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).

Symbols

ProTem has 13 keywords, plus 4 classes of symbols, plus 61 other symbols. Altogether they are:

if then else fi new old for do od result open close unit
number text name comment
% “ ” « » _ ` : :: := = ⧧ < > ≤ ≥ ! ? , ‘ ; . ;.. ,.. | || ( ) { } [ ] ⟨ ⟩
+ – × / ↑ ↓ → ↔ ∧ ∨ @ $^+$ * ~ ⁄ ¢ $ # ∈ ⊆ ∪ ∩ □ Δ ∇ ◃ ▹

Some of the ProTem symbols are not found on standard keyboards. Here are the substitutes.

for “ use "	for ” reuse "	for « use <<	for » use >>
for ⧧ use |=	for ≤ use <=	for ≥ use >=	for ‘ use '
for ⟨ use **par**	for ⟩ use **rap**	for × use &	for ↑ use ^
for ↓ use \	for → use –>	for ↔ use < >	for ∧ use / \
for ∨ use \ /	for $^+$ reuse +	for ⁄ use //	for ¢ use **size**
for ∈ use **elt**	for ⊆ use **sub**	for ∪ use **cup**	for ∩ use **cap**
for □ use []	for Δ use **nand**	for ∇ use **nor**	for ◃ use < |
for ▹ use | >	for “ use ““ or ""	for ” use ”” or ""	

A number is formed as one or more decimal digits, optionally followed by a decimal point and one or more decimal digits, optionally followed by % . Here are six examples.
0   275   27.5   0.21   99%   12.3%
A decimal point must have at least one digit on each side of it.

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. Here are four 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 and continues with any number of letters and digits, except that keywords and symbol substitutes cannot be names. A fancy simple name begins with « , and continues with any number of any characters 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. Here are some examples.
plain simple names: *x* *A1* *george* *refStack*
fancy simple names: «William & Mary» « $x' \ge x$ »
compound names: *ProTem_grammars_Hehner* «2016-9-8»_«grad recruiting»_*DCS*

A comment begins with a ` that is not in a text or fancy name, and ends at the end of a line.

Presentation Grammar

There are 29 ways of forming a program, and 56 ways of expressing data. (An LL($^1/_2$) grammar and an LR($^1/_2$) grammar are at the end of this document.)

A name is one of
- simplename: a simple name (plain or fancy)
- compoundname: more than one simplename joined with underscores

At each point in a program, a simplename is one of
- newname: a simplename that has not previously been defined in the current scope
- oldname: a simplename that has previously been defined in the current scope

At each point in a program, a name is one of
- indvarname: a name defined as an independent variable
- depvarname: a name defined as a dependent variable
- constantname: a name defined as a constant
- channelname: a name defined as a channel
- programname: a name defined as a program or procedure
- dictionaryname: a name defined as a dictionary
- undefinedname: an undefined name

Here are the ways of expressing data. To the right of each there are examples and explanations and pronunciations.

number	0 1.2 10%
+ data	plus, identity
– data	minus, negation, not
data + data	plus, addition
data – data	minus, subtraction
data × data	times, multiplication
data / data	by, division
data ↑ data	to the power, exponentiation
data ∧ data	minimum, conjunction, and
data ∨ data	maximum, disjunction, or
data Δ data	negation of minimum, nand
data ∇ data	negation of maximum, nor
data = data	equals, equation
data ⧣ data	differs from, discrepancy
data < data	less than, strict implication
data > data	greater than, strict reverse implication
data ≤ data	less than or equal to, implication
data ≥ data	greater than or equal to, reverse implication
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	content
data ∈ data	elements of a set
data ⊆ data	subset

data ∪ data	set union
data ∩ data	set intersection
ϟ data	power
$ data	set size
text	“abc”
data ; data	string catenation
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	list length
data $^+$ data	list catenation
data data	list index, function application, composition
data @ data	pointer indexing
⟨ simplename : data → data ⟩	function, create constant parameter
data → data	function, function space
□ data	domain of a function
data | data	selective union
indvarname	independent variable name
depvarname	dependent variable name
constantname	constant name
channelname	the most recent data read on the channel
? channelname	test for presence of input on the channel
if data **then** data **else** data **fi**	conditional data
result simplename : data = data **do** program **od**	programmed data, create local independent variable
(data)	parentheses

Next we have the ways of forming a program.

new newname : data = data	create independent variable : type = initial value
new newname = data	create dependent variable
new newname := data	create constant
new newname **do** program **od**	create program name but don't execute program
new newname ! ? data	create channel with type
new newname **open**	create and open dictionary
new newname **unit**	create constant measuring unit
new newname	forward definition
old oldname	remove or hide
open dictionaryname	open dictionary
close dictionaryname	close dictionary
indvarname := data	assign independent variable to value
channelname ! data	to channel send output
channelname ? data	from channel receive input of this type
channelname ? channelname !	input, correct, and echo
newname **do** program **od**	create program name and execute program
programname	execute (call) named program
⟨ simplename : data → program ⟩	procedure, parameter is constant

⟨ simplename :: data → program ⟩	procedure, parameter is independent variable
⟨ simplename ! data → program ⟩	procedure, parameter is output channel
⟨ simplename ? data → program ⟩	procedure, parameter is input channel
program data	procedure, data argument
program indvarname	procedure, independent variable argument
program channelname	procedure, channel argument
program . program	sequential composition
program ‖ program	parallel composition
if data **then** program **else** program **fi**	conditional program
for simplename := data **do** program **od**	controlled program, create local constant
do program **od**	parentheses

Here is the precedence of the forms of program.

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

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

a **do** *B* **od**. *C*. *D* ‖ *E*. *F*

when fully parenthesized, becomes

do do *a* **do** *B* **od od**. *C*. *D* **od** ‖ **do** *E*. *F* **od**

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

0.	number text name ( ) [ ] { } ⟨ ⟩ **if then else fi** **result do od**	
1.	juxtaposition @	left-to-right
2.	+ – # ~ ⚡ ? □ * → ↑ ↓	prefix + – # ~ ? □ * infix * → ↑ ↓ right-to-left
3.	× / ∩ ∧ ∨ Δ ∇	infix / left-to-right
4.	+ – $^{+}$ ∪	infix – left-to-right
5.	; ;.. ‘	infix
6.	, ,.. | ◁ ▷	infix ◁ ▷ left-to-right
7.	= ⧧ < > ≤ ≥ : ∈ ⊆	infix continuing

On level 7, the operators are "continuing". This means, for example, that $a=b=c$ is neither grouped to the left nor grouped to the right, but means $(a=b)\wedge(b=c)$. Similarly $a<b=c$ means $(a<b)\wedge(b=c)$, and so on.

Whenever "data" appears in an alternative for "program", the most general form of data is intended, with these exceptions: in a variable or parameter declaration, the type must be on precedence level 0; when a function or program is argumented, the argument must be on precedence level 0. Therefore $p\ a\ b$ means $(p\ a)\ b$. Any data expression becomes precedence level 0 by putting it in parentheses ( ) .

Only one alternative for "data" contains "program", and there the most general form of program is intended.

Data

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

Numbers

In addition to the number symbols, there are predefined names of numbers such as *pi* (an approximation to the ratio of a circle's circumference to its diameter), *e* (an approximation to the base of the natural logarithms), and *i* (the imaginary unit, or square root of –1). Predefined names can be redefined in a new scope. In addition to the 1-operand prefix operators + and – , and the 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 Δ is the negation of minimum. The operator ∇ is the negation of maximum.

In ProTem, 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.

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: *backspace* , *tab* , *newline* , *click* , *doubleclick* , and *end* (the end-of-file character). The operators *suc* and *pre* give the successor and predecessor respectively.

Binary Values

There are two predefined binary constants: *true* and *false* . Negation is – , conjunction is ∧ , disjunction is ∨ , nand is Δ , 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 strings of ordered items, and to lists of ordered items, with a binary result. In the binary order *false* is below *true* , so ≤ is implication. The 3-operand **if** *x* **then** *y* **else** *z* **fi** has binary operand *x* , but *y* and *z* are of arbitrary type.

Bunches

There are several predefined bunch names:

null	- empty
nat	- all natural numbers: 0, 1, 2, ...
int	- all integer numbers: ..., –2, –1, 0, 1, 2, ...
rat	- all rational numbers: ..., 1/2, ...
real	- all real numbers: ..., 2↑(1/2), ...
com	- all complex numbers: ..., (–1)↑(1/2), ...

char	- all characters: ..., “a”, ...
bin	- both binary values: *true*, *false*
text	- all texts (character strings): ..., “abc”, ...
pic	- all pictures
all	- all ProTem items

Any number, character, binary value, set, string of elements, and list of elements is an elementary bunch, or synonymously, an element. For example, the number 2 is an elementary bunch, or element. Every 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 $x \le y$. 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, $¢(0, 1, 2) = 3$ and $¢null = 0$.

Bunches are equal if and only if they consist of the same elements, without regard to order or 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 ($nat\uparrow 2$), the natural powers of two ($2\uparrow nat$), 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 ~ . For example, $\sim\{0, 1\} = 0,1$. The size of a set is \$. For examples, $\$\{0, 1\} = 2$ and $\$\{null\} = 0$. The element, subset, union, and intersection operators $\in$ $\subseteq$ $\cup$ $\cap$ are as usual. 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, 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 catenation is denoted by a semi-colon:

    $S ; T$ “ S catenate T ”, “ S join T ”

For example,

    2; 3; 5; 7

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

    x;..y “ x to y ” (same pronunciation as x,..y)

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

The length of a string is obtained by the ↔ operator. For example, ↔(2; 3; 5; 7) = 4 .

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

If S is a string and n is an index of S and i is any item, then $S \triangleleft n \triangleright i$ is a string like S except that item n is i . For example, (3; 5; 9) ◁ 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 catenation 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, ...

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 length, content, indexing, pointer indexing, catenation, 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.

$\# L$	"length of L"
$\sim L$	"content of L"
$L\ n$	"L at n", "L at index n"
$L\ @\ p$	"L at p", "L at pointer p"
$L\ ^+\ M$	"L catenate M", "L join M"
$L\ M$	"L composed with M"
$L \mid M$	"L otherwise M", "the selective union of L and M"

plus the comparisons $L{=}M$, $L{\neq}M$, $L{<}M$, $L{>}M$, $L{\le}M$, $L{\ge}M$.

Here are some examples.

$\#[0;\ 1;\ 2]\ =\ 3$ (the number of items in a list)

$\sim[0;\ 1;\ 2]\ =\ 0;1;2$

$[0;..10]\ 5\ = 5$ (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. 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 $L \mid M$ is indexed with n , the result is either $L\ n$ or $M\ n$ depending on whether n is in the domain $(0,..\#L)$ of L . If it is, the result is $L\ n$, otherwise the result is $M\ n$.

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

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.

Functions

Let p (parameter) be a simple name, let D (domain) be a bunch of items, and let B (body) be an element (possibly using p as a constant name for an element of D). Then

$\langle p\colon D \rightarrow B\rangle$

is a function with parameter p , domain D , and body B . For example,

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

is the successor function on the natural numbers.

A function with two parameters is just a function of one parameter whose body is a function of one parameter. For example, the maximum function is

$\langle a\colon real \rightarrow \langle b\colon real \rightarrow \textbf{if}\ a{>}b\ \textbf{then}\ a\ \textbf{else}\ b\ \textbf{fi}\rangle\rangle$

Similarly for functions with more than two parameters.

The $\square$ operator gives the domain of a function. For example, $\square\langle n\colon nat \rightarrow n{+}1\rangle\ =\ nat$.

The notation for applying a function to an argument is the same as that for indexing a list: juxtaposition. Also, 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 $\langle\ \rangle$ and unused name. For example,

$2\to3$

abbreviates $\langle n\colon 2 \to 3\rangle$ or choose any other parameter name. An example of its use is

$1\to21 \mid [10;\ 11;\ 12]\ =\ [10;\ 21;\ 12]$

We allow domains to be strings in the following circumstances.

$nil\to x \mid f\ =\ x$

$(x;y) \to z \mid f\ =\ x\to(y\to z \mid f\,x) \mid f$

Thus, for example,

$$\begin{aligned}(0;1) \to 6 \mid [[0;\ 1;\ 2];\\ [3;\ 4;\ 5]]\ =\ [[0;\ 6;\ 2];\\ [3;\ 4;\ 5]]\end{aligned}$$

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

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

Allowing the body of a function to be a bunch generalizes the function to a relation. For example, $nat\to bin$ can be viewed in either of the following two equivalent 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

$\langle n\colon nat \to mod\ n\ 2 = 0\rangle :\ nat\to bin$

Programmed Data

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

First, a local independent variable is introduced with a type and initial value; its scope is from **do** to **od** . Then the program is executed. The result is the final value of the newly introduced 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 local scope; their values may be used, but assignments to them are not permitted. Input and output are not permitted.

Names and Dictionaries

Each name in a dictionary is defined to be one of the following: an independent variable, a dependent variable, a constant, a program, a channel, or a dictionary. When a name is defined to be a dictionary, this dictionary also can contain names, some of which can be defined as dictionaries, and so on. Therefore there is a tree of dictionaries. Whether this tree has a root, and if so what its name is, are of no consequence. Suppose there is a text named *ProTem* within a dictionary named *grammars* within a dictionary named *Hehner* within a dictionary named *cs* within a dictionary named *utoronto* within a dictionary named *ca* . This text can be referred to as *ProTem_grammars_Hehner_cs_utoronto_ca* .

A dictionary is either closed or open. We can **open** a closed dictionary, and **close** an open dictionary. By opening dictionaries, we can shorten the names we use. The text referred to by the lengthy compound name in the previous paragraph can be referred to simply as *ProTem* if the dictionary *grammars* is open. The predefined names include a dictionary named *complex* , within which there is a name *i* . It can always be referred to as *i_complex* . If we are going to refer to it often, we might want to shorten this. We do so by saying **open** *complex* , and then we can say just *i* .

Names are defined in a variety of ways, including as **new** , as function parameters, as procedure parameters, as **for**-loop parameters, and as **result** variables. Whenever a name is defined, its definition is written in the open dictionary that was opened last (there is always one such dictionary, even initially, though its name may not be known). A name being defined must not already be defined in the current scope (see **Scope**, later). But it may already be in the open dictionary that was opened last; in that case, the new definition replaces the old definition until the end of the current scope or until it is removed by **old** .

Whenever a simple name is used, it is looked up in the open dictionary that was opened last (there is always one such dictionary, even initially, though its name may not be known); if it is not there, it is looked up in the open dictionary that was opened next-to-last; and so on. The first definition found for the name is the one used. If the name is not in any open dictionary, it is unknown (even though it may be in some closed dictionaries).

Whenever a compound name is used, it is looked up as follows. The last simple name in the compound name is looked up in the usual way (starting with the open dictionary that was opened last). Its definition must be as a dictionary. The simple name before the last one in the compound name is looked up in this one dictionary (whether open or closed). And so on for preceding names in a compound name.

Names defined by **new** can be removed from a dictionary with the keyword **old** (it must already be there). Names are also removed from a dictionary when execution exits the right scope bracket of the scope in which they were introduced. Further details and examples will be presented later (see **Scope**).

Programs

A fifth of the program constructs are concerned with dictionaries: adding names (**new**), deleting names (**old**), opening a dictionary (**open**), and closing a dictionary (**close**). The other four-fifths are variable assignment, input, output, and a variety of ways of combining programs to form larger programs. All programs, including those that add or remove names from a dictionary, including those that open or close a dictionary, are executed in their turn, just like variable assignments and input and output.

Independent Variable Definition

Here is an example independent variable definition (declaration).

new x: $nat = 5$

This defines x to be an independent 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 an independent variable definition, the data after the colon is called the "type" of the variable. The type can be anything except the empty bunch. The type and initial value can depend on previously defined names, including variables. For example,

new y: $(0,..2\times x) = x$

defines y as an independent 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* . 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 an independent 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 an independent variable that can be assigned to any text of length less than 20, and is initially assigned to the text “abc” .

Assignment

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

x:= 6
s:= 3 → 5 | *s*

The data on the right must be an element in the type of the variable on the left.

Dependent Variable Definition

If independent variable *x* is defined as

new *x*: *nat* = 5

then

new *xplus1* = *x*+1
new *xplus2* = *xplus1*+1

make *xplus1* and *xplus2* dependent variables. They depend on variable *x* , so that *xplus1* = *x*+1 and *xplus2* = *xplus1*+1 are always true. Independent variable *x* can be assigned various values. But dependent variables *xplus1* and *xplus2* cannot be assigned; their values change when the value of *x* changes. Expressions *x*+1 and *xplus1*+1 are not evaluated in the definition; they are evaluated each time *xplus1* and *xplus2* are used. (A clever implementation will evaluate, at definition time, all parts of the expression that do not depend on variables, and will re-evaluate *xplus1* and *xplus2* only when *x* may have changed value.)

Constant Definition

Here are three constant definitions.

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

where *pi* is a predefined constant in dictionary *calculus* .

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

new *xplus3*:= *x*+3

In the dependent variable definition of *xpus1* earlier, *x*+1 is not evaluated at definition time; it is evaluated every time *xplus1* is used. By contrast, the constant definition *xplus3* evaluates *x*+3 once, at definition time.

When there are no variables used to express the value, there is no semantic difference between dependent variable definition and constant definition, but there may be an efficiency difference.

Data Recursion

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

new *no*: $(0,..2\times no) = no$

is not allowed due to the occurrences of *no* to the right of the colon. Likewise a constant definition cannot be recursive.

Dependent variable 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* $= 0 \to 1 \mid \langle n\text{: } (nat+1) \to n \times fact\ (n-1)\rangle$

new *div* $= \langle\ a\text{: } nat \to \langle d\text{: } (nat+1) \to$
if $a<d$ **then** 0 **else if** *even a* **then** $2 \times div\ (a/2)\ d$ **else** $1 + div\ (a-d)\ d$ **fi fi**$\rangle\rangle$

Here is a function that eats arguments until it is fed argument 0 .

new *eat* $= \langle a\text{: } nat \to$ **if** a=0 **then** 0 **else** *eat* **fi**$\rangle$

So *eat* 5 2 0 = 0 and *eat* 4 7 3 8 0 = 0 .

The next example is a pure, baseless recursion.

new *rec* = *rec*

Whenever *rec* is used, the computation will be nonterminating.

A final example defines all binary trees with integer nodes.

new *tree* = [*nil*], [*tree*; *int*; *tree*]

Program Definition

Program definition gives a program a name, but does not execute the program. For example,

new *switchends* **do** s:= $0 \to s\ 9 \mid 9 \to s\ 0 \mid s$ **od**

Execution of this definition creates the program name *switchends* , but does not execute program *switchends* . After execution of this definition, the name *switchends* can be used to cause execution of the program it names. Program definitions can be recursive.

The names used in a program definition, in the previous example *s* , are those visible at the time the definition is executed, that is, at the time this definition adds the name *switchends* to the dictionary. At the time *switchends* is called, causing execution of the assignment of *s* , variable *s* may not be visible, but it is assigned nonetheless.

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 real number constant. So, for example, we write $10\times m/s$ for the speed 10 meters per second. And we can define

new *km*:= $1000\times m$

to make *km* be a kilometer, and

new *h*:= $3600\times s$

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

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

assigns *speed* to $1\times m/s$. For another example,

new *sheet* **unit**. **new** *quire*:= 25×*sheet*. **new** *ream*:= 20×*quire*.
new *order*: (*nat*×*sheet*) = 3×*ream*

assigns *order* to 1500×*sheet* . When the value $5\times m/s$ is converted to text by *realtext* , the result is "5 *m*/*s*" without the × sign and without evaluating the unknown real values *m* and *s* .

Forward Definition

A forward definition, for example

new *abc*

is a notice that a definition will follow later. It is used, for example, when definitions are mutually recursive. (See **Scope**.)

Name Removal

Names added to a dictionary with the keyword **new** can be removed from the dictionary with the keyword **old** . Even though a name may be removed from a dictionary, its definition 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 for everyone's use. The name *s* was defined for the purpose of defining the other names, and then removed from the dictionary, leaving the other names dependent upon an anonymous variable.

Dictionaries

The syntax

new *d* **open**

is used to create a new dictionary, entering its name *d* in the open dictionary that was opened last, and then opening *d* . The syntax

open *d*

is used to open an existing but closed dictionary *d* . The syntax

close *d*

is used to close an existing open dictionary.

The predefined names include a dictionary named *randomnat* , within which there are three names: *init* , *next* , and *value* . It might have been defined as:

```
new randomnat open.
    new big:= 2↑31.
    new rv: (0,..big) = 123456789.
    new init do ⟨seed: (0,..big) → rv:= seed⟩ od.
    new next do rv:= mod (rv × 5↑13) big od.
    new value = ⟨from: nat → ⟨to: nat → floor (from + (to–from)×rv/big)⟩⟩.
    old big.  old rv.
close randomnat.
```

Variable *rv* is now hidden; its name is removed from the dictionary, but *init* , *next* , and *value* still use it. We can use the definitions in this dictionary in the following way:

```
init_randomnat 555555555.
next_randomnat.
screen! nattext (value_randomnat 0 10).
```

Or, if we are going to use them often, we may want to shorten what we say as follows:

```
open randomnat.
init 555555555.
next.
screen! nattext (value 0 10).
```

We can get rid of a dictionary name *d* by saying

```
old d
```

Removing a dictionary name by **old** also removes all names in that dictionary. The dictionary remains in existence, closed and anonymous, as long as something refers to it or to its contents.

Sequential Composition

Sequential composition is denoted by a period. It is an infix connective.

Parallel Composition

For programs P, Q, ..., R that each assign different variables, or different parts of a structured variable, their parallel composition is denoted $P \| Q \| ... \| R$. Each program can use the independent variables assigned by the others, but all occurrences of independent variables assigned by the other programs refer to their initial value. Similarly a dependent variable that depends on variables assigned in one program can be used in parallel programs, but its value will be determined by the initial values of the variables it depends on. 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.

Here is a program to find the maximum value in nonempty list L in $log\ (\#L)$ time. (L is an independent variable, and its 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 .

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

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

Output and Input

Each channel is defined to transmit a specific type of value. The output channels *screen* and *printer* , and the input channel *keys* , are predefined to transmit text.

Channel *screen* accepts text, which is displayed on the screen. The program
screen! "Hi there."
sends the text "Hi there." to the screen. A string of outputs can be sent together
screen! "Answer = "; *realtext x*; *newline*
where *realtext* converts from a real number to a text.

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. So the shift-A combination and the control-Q combination are characters. The click button is just a key like any other; *click* and *doubleclick* 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*
If input is not yet available, it is awaited. The *backspace* and *newline* characters may be part of the input; no corrections are made. The input is not echoed on the screen. The program
keys? *text*; *newline*
reads text up to and including a *newline* character. To receive spaces followed by digits, possibly including a decimal point, define
new *digits*:= "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
and then write
keys? *" "; *(*digits*, ".")
The longest matching string of digits and decimal points will be read.

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

There is a second form of input that replaces the type with the name of a channel that outputs text.
keys? *screen* !
reads text from channel *keys* , corrected according to *backspace* characters, up to the next *newline* character, and echoes the input on the screen. The *newline* character is consumed and echoed, but not included in the value of *keys* .

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* .

Channel Definition

The definition
new *c*!? *nat*
defines *c* to be a new local channel that transmits naturals. It can be used for output and input. For example,

```
    new c!? nat.  do c! 7 || c? 0,..100.  x:= c od.  old c
```
assigns x to 7 . Only one of the programs that are in parallel with each other can use a channel for output. More than one of the programs that are in parallel with each other 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 **if then else fi** is as usual. There is no one-tailed **if** in ProTem, but there is a predefined program *ok* whose execution does nothing. For example,

```
    if x>y then x:= y else ok fi
```
With a one-tailed **if**, it is too easily forgotten that there are two cases to consider. An “assert” program is obtained according to the following example.

```
    if x>y then ok else screen! “appropriate error message”.  stop fi
```

Named Programs

A named program has the syntax

```
    newname do program od
```
The name is attached to the program (like a program definition), and the program is executed (unlike a program definition). The program name is known only within the program to which it is attached; after that, it is again new and can be reused. One purpose of this naming is to make loops. Here is a two-dimensional search for x in an $n{\times}m$ array A of integers (that is, A: [n*[m*int]]).

```
    new i: nat = 0.
    tryThisI do  if i=n then screen! inttext 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 screen! inttext x; “ occurs at ”; nattext i; “ ”; nattext 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 natural d , using only addition and subtraction.

```
    r:= a.
    outerloop do  if r<d then ok
                  else 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. This example means the same as

```
r:= a.
outerloop
    do if r<d then ok
        else new dd: nat = d.
              innerloop
                  do r:= r–dd.  dd:= dd+dd.
                      if r<dd
                      then if r<d then ok
                             else new dd: nat = d.
                                   innerloop
                                       do r:= r–dd.  dd:= dd+dd.
                                           if r<dd then outerloop else innerloop fi od fi
                      else  r:= r–dd.  dd:= dd+dd.
                             if r<dd then outerloop else innerloop fi 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 $x:=f_n$ and $y:=f_{n+1}$, where f_0 , f_1 , f_2 , ... are the Fibonacci numbers, 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↑2 + y↑2.  y:= 2×n×y + y↑2
                  else n:= n/2 – 1.  Fib.  n:= x.  x:= 2×x×y + y↑2.  y:= n↑2 + y↑2 + x 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 $x:= x+1$. 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=0 then ok else x:= x–1.  « x′ = 0 » fi od
```

In the **else**-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.

The following three lines are equivalent to each other.

```
P do Q od
new P do Q od.  P.  old P
do new P do Q od.  P od
```

Controlled Program

This example computes the transitive closure of A: $[n*[n*bin]]$.

```
for j:= 0;..n
do  for i:= 0;..n
      do  for k:= 0;..n
            do  A:= (i;k) → (A i k  ∨  (A i j ∧ A j k))  |  A od od od
```

The assignment can be restated as

if $A\ i\ j \wedge A\ j\ k$ **then** $A{:=}\ (i;k) \rightarrow true \mid A$ **else** *ok* **fi**

if you prefer. The name being introduced by **for** is known only within the loop body, and it is known there as a constant. It is not a variable, and so it is not assignable. We call it a **for** parameter. In the example, each parameter 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 $prime$: $[n{*}bin] = [2{*}false;\ (n-2){*}true]$.
for $i{:=}\ 2;..ceil\ (sqrt\ n)$
do **if** $prime\ i$ **then for** $j{:=}\ i;..ceil\ (n/i)$ **do** $prime{:=}\ (i{\times}j) \rightarrow false \mid prime$ **od else** *ok* **fi od**

A **for** parameter is "by initial value", so

for $i{:=}\ x;\ x$ **do** $x{:=}\ i+1$ **od**

increases x by 1 , not 2 .

After the := we can have any string expression; the parameter stands for each item in the string, in sequence. We can also have any bunch expression; the parameter stands for each element of the bunch, in parallel. As an example,

for $i{:=}\ 0,..\#A$ **do** $A{:=}\ i \rightarrow 0 \mid A$ **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.)

Procedures

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

$\langle y{:}\ real \rightarrow x{:=}\ x{\times}y\rangle$

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. For example,

$\langle y{:}\ real \rightarrow x{:=}\ x{\times}y\rangle\ 3$

which is the same as

$x{:=}\ x{\times}3$

A procedure's constant parameter is known only within the procedure body. It is not a variable, and so it is not assignable. It is "by initial value", so

$\langle i{:}\ int \rightarrow x{:=}\ i.\ y{:=}\ i\rangle\ (x+1)$

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

A program can also have an independent variable parameter, as in this example.

$\langle x{::}\ int \rightarrow x{:=}\ 3\rangle$

A procedure with an independent variable parameter cannot be applied to a variable appearing in the procedure. This example procedure can be applied to any independent variable, even one named x , because the nonlocal name x does not (and cannot) appear in the procedure. The procedure

$\langle x{::}\ int \rightarrow x{:=}\ 3.\ \ y{:=}\ 4\rangle$

cannot be applied to variable y . The main use for independent 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.

$\langle c!\ text \rightarrow c!$ “abc”$\rangle$

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,

$\langle c?\ text \rightarrow c?.\ screen!\ c\rangle$

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

$\langle c!\ text \rightarrow c!$ “abc”. $d!$ “def”$\rangle$

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

Format

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

```
        A. B
or
        A.
        B
-----------------------------------
        A || B
or
          A
        || B
-----------------------------------
        if A then B else C fi
or
        if A then B
        else C fi
or
        if A
        then B
        else C fi
```

```
        for x:= A do B od
or
        for x:= A
        do B od
-----------------------------------
        A + B
or
          A
        + B
-----------------------------------
        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⟩⟩
```

Scope

Scopes are limited by **do od** , **then else** , **else fi** , and $\langle\ \rangle$ brackets. Each of these four pairs is a scope opener and a scope closer. Scopes are also limited by parallel composition; $||$ is both a scope closer and a scope opener.

A name introduced by the keyword **new** must be new, i.e. not defined since the previous unclosed 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 definition in a more local scope. For example, letting $A, B, C, ...$ stand for arbitrary program forms (but not **new** or **old**), in

A. **new** x: *int* $= 0$. B. **do** C. **new** x: *bin* $=$ *true*. D **od**. E

the definition of x as an integer variable is not yet in effect in A , but it is in effect in B , C , and E . The definition that makes x a binary variable is in effect in D . None of A , B , C , D , or E can contain a redefinition of x unless it is within further **do od** , **then else** , **else fi** , or $\langle\ \rangle$ brackets.

A name introduced by **new** can be removed from the dictionary by using **old** , ending its scope early. So in

new $x := 0$. 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 previous unclosed scope opener. After **old** x , the name x is again new and available for definition. However,

new $x := 0$. **do old** x. A **od**

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

If a name is introduced by **new** outside all scope limiters, 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 declared outside all scope limiters serve as “files”. A predefined name cannot have its scope ended by **old** , but it can be obscured by a programmer's redefinition of the same name.

In an independent variable definition, a constant definition, a channel definition, a **for** parameter definition, a function parameter definition, a procedure parameter definition, and a **result** variable definition, the name being introduced cannot be used in the type or initial value; its scope begins after the type and initial value.

In a dependent variable or program definition, the scope of the name being introduced starts immediately. This allows the definitions to be recursive. A forward definition allows mutual recursion by starting the scope of a dependent variable name or program name even before its definition. For example, in

new $f := 3$. **do new** f. **new** $g = \cdots f \cdots g \cdots$. **new** $f = \cdots f \cdots g \cdots$. B **od**

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** $f := 3$.

A program can be given a name without the keyword **new** . Any such name must be new within the most local scope, just like a name introduced with the keyword **new** . Its scope extends only through the program to which it is attached, not beyond. After that, it is again new and available for definition.

A name can be introduced as a procedure parameter or function parameter or **for** parameter or **result** variable. Any such name is automatically considered to be new. Its scope extends only through the program or data to which it is attached, not beyond.

The opening and closing of dictionaries obey the same scope rules. In a program of the form

    A. **do** *B* **od**. *C*

all names in all dictionaries, and which dictionaries are open, and the order in which they were opened, are the same at the start of *C* as they were at the end of *A* , regardless of any local changes within *B* . However,

    open *d*. **do close** *d* **od**

is not allowed; a dictionary cannot be closed in a subscope of the one in which it was opened.

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

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 program can be presented to a compiler as a text. But we cannot write a self-reproducing expression with this convention. For that purpose, we would need to represent left- and right-double-quote characters within a text by repeating them. For example, “Just say ““no””.” .

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

    screen! *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 catenate new records onto its end.

    file:= *file*; *p*

The efficiency of pointers is obtained through the use of predefined name *index* .

    new *index*:= *text*→**nat*

When applied to a text argument, it yields the result **nat* . The use of *index* is a signal to the implementation that the natural numbers will be used only as indexes into the structure whose name is given by the text argument (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” → “zzzzz” | “next” → 0].
    new *first*: *index* “*G*” = 0.

We can use *first* in an arithmetic context, 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

and similarly to move it down to the right. Thus p is a string of indexes indicating a subtree *t*@*p* of t. We can replace this subtree with tree s using the assignment

t:= *p* → *s* | *t*

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

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

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

t:= (*p*;1) → 6 | *t*

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

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

Now

p:= *backup p*

moves p up to its parent.

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

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

Here is a procedure with one parameter

new *translateX* **do** *transformX* 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 *magnifyX* **do** ⟨*magnification*: *real* → *transformX magnification* 0⟩ **od**

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

magnifyX 3
transformX 3 0

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

new *fact* = ⟨*n*: *nat* → **result** *f*: *nat* = 1**do** **for** *i*:= 0;..*n* **do** *f*:= *f*×(*i*+1) **od** **od**⟩

Exception handling is provided by bunch union or by the | operator. 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? formnum.  P
                        else if ?d then d? formnum.  Q
                             else inputchoice fi fi od
```

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 newlines from text, and *sort* is a procedure to sort texts. (Please excuse the informal body since it's not the point of the example.)

```
        new trim do ⟨in? text → ⟨out! text → repeatedly read from in , trim off leading and trailing
                                              space, output to out , until "***" is read.
                                              The final "***" is output ⟩⟩ od.
        new sort do ⟨in? text → ⟨out! text → repeatedly read from in until "***" is read and output
                                              the sorted texts to out . The final "***" is output ⟩⟩ 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.  do trim keys pipe || sort pipe screen od.  old pipe
```

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

The effect of modules is partly obtained by **old** and partly by dictionaries. There is no direct counterpart to the import construct or frame construct. It is recommended to place a comment at the head of each major program component saying which nonlocal names are used, and in what way they are used. It is possible for an implementation to generate such comments on request. It is also possible for programmers to make such comments in an agreed format so that an implementation can recognize them and check them. Here is a suggested standard.

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

They are transitive through "need" and "call" without requiring the implementation to do a transitive closure (it just checks the comments at the head of the needed constants and called program names).

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.

Object Orientation

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, and should be done whenever it is helpful. 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 and used many times. Or, if you prefer, objects can be instantiated by re-invoking the program that defines one of them.

Documents

The predefined name *pic* is all picture values. It can be used, for example, to create a picture-valued variable.

```
    new p: pic = [x*[y*0]].
```

The name *pic* is defined as [*x**[*y**(0,..*z*)]] where *x* is the number of screen 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 .

Another predefined name is *movie* , defined as [**pic*] . The operations on movies are just those of lists, such as catenation. 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.

Editing

The command control-e (hold down the control key and type an e) invokes an editor for creating or modifying any definition (independent variable, dependent variable, constant, program name, channel, or dictionary name). When a program name is defined, the defined program is not immediately compiled; it is compiled when it is first invoked. When its definition is modified, the old executable form is thrown away; the new definition is not compiled until it is invoked. It may also be necessary to throw away the executable form of all programs that depend directly on the redefined name.

Security

Any dictionary may contain a data definition of the name *password* , such as

```
    new password:= encode “Smith” ` my mother's maiden name
```

where *encode* is a not-easily-invertible function from texts to texts. If a dictionary contains the constant *password*, the text will be requested when an attempt is made to open the dictionary or to refer to its contents. Passwords belong to dictionaries, not to people. For example

```
    new readBarrier open.
        new password:= encode “elephant”. ` code for reading
        new writeBarrier open.
            new password:= encode “giraffe”. ` code for writing
            new it: real = 17.2.
            close writeBarrier.
        new readonlyit = it_writeBarrier.
        close readBarrier.
```

To use *readonlyit* , either by opening dictionary *readBarrier* or as *readonlyit_readBarrier* , you must know the password “elephant”. This enables you to know the value of variable *it* , but not to change it. To change it, you must know a second password, “giraffe”.

Session

When the computer is turned on, a session begins. When control-q is typed, a session ends and a new one begins. When a number of idle minutes pass (the number is a parameter of the system and may be set to infinity), a session ends and a new one begins. When the computer is turned off, a session ends.

At the start of a session, the screen is clear, only the root dictionary is open, and all passwords are required. A password will not be requested twice within the same session for the same dictionary.

Sessions do not define the lifetime of definitions (variables, data, programs, dictionaries). A definition that is outside all **do od** , **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 nonlocal variables whose values are stored in volatile memory, and whose values outlast a session, to be saved in permanent memory.

Sessions are defined for each user of a multiuser computer, and are for security and error recovery.

Error Recovery

It is essential to be able to abort the execution of a program, especially if you suspect that its execution will take forever. To do so, type control-u (for “undo”). The undo command not only aborts execution, but also returns to the state (except for input and output) prior to the start of execution of the aborted program. The undo command can even be issued after the completion of execution of a program, before the start of the next one. In that case it acts as the magical inverse of the previous program.

On many computers, undo can be implemented just by doing nothing; nonvolatile memory contains the state as it was before the start of the previous program, and volatile memory contains the current state, which is stored in nonvolatile memory at the start of execution of the next program. (When the execution of a program runs over five minutes, or causes a massive state change, the current state may be saved temporarily in nonvolatile memory, to become permanent when the possibility of undoing it has passed.)

A second level of error recovery, control-s, undoes a session. Implementing it requires capturing the state at the start of a session. Although this is expensive, it is hoped that it can serve also as system backup, performed automatically and incrementally with a frequency that matches file use.

The final kind of error recovery works in conjunction with session undo. It requires ProTem to keep a text file named *session* consisting of all keystrokes since the start of the session. (This is quite practical: an hour's hard work produces only 10kbytes of keystrokes.) One first performs a session undo; this resets the state except for the keystroke file. One then makes a copy of the keystroke file to capture it at some instant (it is always growing).

new *copy*: *text* = *session*

One then edits the copy, perhaps using the text editor, and then executes the result.

exec copy

This gives us perfectly flexible error recovery for the modest cost of a keystroke file.

Command Summary

There are four “commands” in ProTem that are not presented in the grammar. They cannot be part of a stored program. They can be used only by a human at a keyboard. They are:

control-e:	enter editor
control-q:	quit session
control-u:	undo program
control-s	undo session

Possibly Needed, But Not Yet Designed Features

We need to be able to easily express the creation, deletion, placement, movement, resizing, and scrolling of a window, and to replace any region within a window. The entire screen, sometimes called the “desktop”, is just a window that cannot be created (it is already created), deleted, moved, resized, or scrolled. Perhaps we also need better ways of defining touchpad or touchscreen gestures. The variable *cursor*: *nat*; *nat* tells the current cursor position.

We need a sound (noise) data type. We also need a way to combine all of these types in one document. We also need to be able to define regions of documents to be clickable links.

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.

one-tailed **if**

if *x*>0 **then** *y*:= 5 **fi**	abbreviates	**if** *x*>0 **then** *y*:= 5 **else** *ok* **fi**

assertion

assert *x*>*y*	abbreviates	**if** *x*>*y* **then** *ok* **else** *screen*! “assert failure”. *stop* **fi**

list item assignment

A 3:= 5	abbreviates	*A*:= 3→5 | *A*
A 3 4:= 5	abbreviates	*A*:= (3;4)→5 | *A*

definition grouping

new *x*, *y*: *int* = 0	abbreviates	**new** *x*: *int* = 0. **new** *y*: *int* = 0
old *x*, *y*	abbreviates	**old** *x*. **old** *y*
open *this*, *that*	abbreviates	**open** *this*. **open** *that*
⟨*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*⟩⟩

looping constructs

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

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

loop *P*. **exit when** *n*=0. *Q* **pool** abbreviates
loop **do** *P*. **if** *n*=0 **then** *ok* **else** *Q*. *loop* **fi** **od**

Implementation Philosophy

Ideally, an implementation checks whether the text presented to it is a program, and issues an error message if it is not. That check should include determining whether every independent 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 be evaluated further. That presents an implementation problem, but not a semantic problem. For example,

screen! *inttext* (–3)	prints –3

We cannot evaluate the application of the minus operator to the number 3 , so the implementation prints the operator and operand. Similarly

screen! *rattext* (1/0)	should print 1/0
screen! *nattext* ([0; 1] 2)	should print [0; 1] 2
screen! *nattext* (⟨*r*: *rat* → 5⟩ (1/0))	should print 5
screen! *bintext* (1/0 = 1/0)	should print *true*
screen! *bintext* ([0; 1] 2 = [0; 1] 2)	should print *true*

No general-purpose programming language has ever been, or will ever be, implemented entirely. Every such 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 "Sorry: this implementation is too limited to accommodate your program.". An "error" message tells a programmer to correct the error; there is no other option. A "sorry" 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 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 a "sorry" 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 declaration, 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. 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, so the next implementation can implement programs that previous implementations could not accommodate.

Predefined Names

abs: *real*→*real*. Absolute value. $abs\ x = $ **if** $x \ge 0$ **then** x **else** $-x$ **fi**

all. All ProTem items.

asm. A machine-dependent program with one text parameter. If the argument represents an assembly-language program, the execution is that of the represented assembly-language program.

await. A program with one parameter of type $real \times s$. 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. See *time* and *wait* and s .

backspace: *char*.

backup: **nat* → **nat*. $backup\ (s; i) = s$.

$bin\ =\ true, false$.

bintext: *bin*→*text*. *bintext true* = “true” and *bintext false* = “false”.

calculus. A dictionary containing the following names.

$e = 2.718281828459045$ (approx). An approximation to the base of the natural logarithms.

exp: *com*→*com*. An approximation to $e \uparrow x$.

lb: $\S\langle r: real \to r>0\rangle \to real$. An approximation to the binary logarithm (base 2).

ln: $\S\langle r: real \to r>0\rangle \to real$. An approximation to the natural logarithm (base e).

log: $\S\langle r: real \to r>0\rangle \to real$. An approximation to the common logarithm (base 10).

$pi = 3.141592653589793$ (approximately). An approximation to the ratio of a circle's circumference to its diameter.

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

char. The characters.

charnat: *char*→*nat*. A one-to-one function with inverse *natchar* .

click: *char*.

com. The complex numbers.

complex. A dictionary containing the following names.

arc: $com \to \S\langle r: real \to 0 \le r < 2 \times pi\rangle$. An approximation to the angle or arc of a complex number.

$i = sqrt\ (-1)$. The imaginary unit.

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

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

abs: *com*→*real*. Absolute value. $abs\ x\ =\ sqrt\ (re\ x \uparrow 2 + im\ x \uparrow 2)$.

comtext: *com*→*text* A text representation of a complex number.

cursor: *nat*; *nat*. A variable telling the current cursor position.

dictionary: *text*. A readable summary of the content of the open dictionary that was opened last.

div: $real \to \S\langle r: real \to r>0\rangle \to int$. $div\ a\ d$ 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*.

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

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

eval: *text*→**all*. 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.

even: *int*→*bin*.

exec. 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. In *exec* "*x*:= *x*+1" , the "*x*" refers to whatever *x* refers to at the location where *exec* "*x*" occurs.

false: *bin*. A binary value.

find: *all*→[**all*]→*nat*. If *i* is an item in *L* , then *find i L* is the index of its first occurrence; if not, then *find i L* = #*L* .

fit: *text*→*int*→*text*. If $i \ge 0$ then *fit t i* 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 $i \le 0$ then *fit t i* 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*. *floor r* ≤ *r* < 1 + *floor r*

form: *real*→*nat*→*nat*→(*nat*+1)→*text*. Format a real number. *form r d e w* 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 pi 4 1 12 = " 3.1416×10↑0" . *form* (–*pi*) 2 0 6 = " –3.14" .
form 5 0 0 3 = " 5" . *form* (–5) 0 0 3 = " –5" . *form* 123 0 0 2 = "**" .

formnum. A format for reading a number from a channel.

g **unit**. A unit representing mass in grams.

hyperbolic. A dictionary containing the following names.
cosh: *com*→*com*. An approximation to a hyperbolic function.
sinh: *com*→*com*. An approximation to a hyperbolic function.
tanh: *com*→*com*. An approximation to a hyperbolic function.

index = *text*→**nat*. A signal to the implementation that the string will be used only as an index to the indicated structure.

int. The integers.

inttext. A text representation of an integer number.

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

m **unit**. A unit representing distance in meters.

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

mailout!? *text*. 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*. 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*. The maximum representable integer (machine dependent).

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

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

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

movie = **pic*.

nat. The natural numbers.

nattext. A text representation of a natural number.
natchar: *charnat char* → *char*. A one-to-one function with inverse *charnat* .
newline: *char*. The return or newline character.
nil. The empty string.
null. The empty bunch.
odd: *int*→*bin*.
ok. A program whose execution does nothing.
openlist: *text*. The names of the open dictionaries in the order they were opened.
pic = [*x**[*y**(0,..*z*)]] where *x* is the number of screen pixels in the horizontal dimension, *y* is the number in the vertical dimension, and *z* is the number of pixel values. The screen pictures.
pre: *char*→*char*. The predecessor function.
printer!? *text*. To the printer, it is an input channel; to all other programs, it is an output channel.
randomnat. A dictionary containing the following three names.
  init. A program with one natural parameter. Its execution assigns a hidden variable to the natural value.
  next. A program. Its execution assigns the hidden variable to the next value in a random sequence.
  value: *nat*→*nat*→*nat*. A reasonably uniform function, dependent on the hidden variable, over the interval from (including) the first argument to (excluding) the second argument.
randomreal. A dictionary containing the following three names.
  init. A program with one real parameter. Its execution assigns a hidden variable to the real value.
  next. A program. Its execution assigns the hidden variable to the next value in a random sequence.
  value: *real*→*real*→*real*. A reasonably uniform function, dependent on the hidden variable, over the interval between the arguments.
rat. The rational numbers.
rattext. A rext representation of a rational number.
real. The real numbers.
realtext: *real*→*text* A text representation of a real number.
round: *real*→*int*. r–0.5 ≤ *round* r < r+0.5
s **unit**. A unit representing time in seconds.
screen!? *text*. To the screen, it is an input channel; to all other programs, it is an output channel.
session: *text*. A text expression giving all keystrokes on channel *keys* since the start of a session.
sign: *real* → (–1, 0, 1).
sort: **ord*→**ord* where *ord* = *real*, *char*, [**ord*].
sqrt: *com*→*com*. An approximation to the principle square root.
stop. A program whose execution does nothing and takes forever so that no computation can follow.
subst: *all*→*all*→**all*→**all*. *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*. The successor function.
tab: *char*.
text = **char*.
textcom: *text*→*com*. If the argument represents a complex number, the result is the represented number.
textint: *text*→*int*. If the argument represents an integer, the result is the represented number.
textnat: *text*→*nat*. If the argument represents a natural number, the result is the represented number.
textrat: *text*→*rat*. If the argument represents a rational number, the result is the represented number.
textreal: *text*→*real*. If the argument represents a real number, the result is the represented number.

texttime: *text*→(*int*×*s*). If the argument represents a time, the result is the represented time in seconds since or before 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0). For example
texttime "1947 September 16 at 19:24 UTC" = –68675760×*s* .

time!? *real*×*s*. 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). Times before then are negative.

timetext: (*real*×*s*)→*text*. A readable form of the time in seconds since or before 2000 January 1 at 0:00 UTC (the midnight that begins 2000 January 1 at longitude 0). For example,
timetext (–68675760×*s*) = "1947 September 16 at 19:24 UTC"

trig. A dictionary containing the following names.

arccos: $§\langle r\text{: } real \rightarrow -1 \le r \le +1\rangle \rightarrow §\langle r\text{: } real \rightarrow 0 < r < pi/2\rangle$. An approximation to a trigonometric function.

arcsin: $§\langle r\text{: } real \rightarrow -1 \le r \le +1\rangle \rightarrow §\langle r\text{: } real \rightarrow 0 < r < pi/2\rangle$. An approximation to a trigonometric function.

arctan: $real \rightarrow §\langle r\text{: } real \rightarrow 0 < r < pi/2\rangle$. An approximation to a trigonometric function.

cos: $real \rightarrow §\langle r\text{: } real \rightarrow -1 \le r \le +1\rangle$. An approximation to a trigonometric function.

sin: $real \rightarrow §\langle r\text{: } real \rightarrow -1 \le r \le +1\rangle$. An approximation to a trigonometric function.

tan: $(§\langle r\text{: } real\cdot \neg\exists\langle i\text{: } int\cdot r = (2\times i + 1)\times pi\rangle\rangle) \rightarrow real$. An approximation to a trigonometric function.

trim: *text*→*text*. A text formed from the argument by removing all leading and trailing *space* , *tab*, and *newline* characters.

true: *bin*. A binary value.

wait. A program with one parameter of type *real*×*s* . If the argument is nonnegative, its execution does nothing but takes the length of time in seconds given by the argument. If the argument is nonpositive, its execution does nothing. See *await* and *time* and *s* .

Example Program

```
new simport ` a program to simulate portation
do `input: keys time
   `output: screen
   `need: ceil index nat real rat sqrt newline nattext textnat m s
   `call: stop wait
   `refer: randomnat

   ` Distance between control boxes is always 1 m.
   ` Merges do not overlap, so 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 declarations

   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
                         | (“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.
             ` 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 declarations, start of initialization

new x: nat = 0. `for input
for b:= 0;..numboxes
do  screen! “What box is ahead-left of box ”; nattext b; “? ”.
    keys? screen!.  x:= textnat keys.
    box:= (b; “ahead left”) → x | (x; “behind left”) → b | box.
    screen! “What box is ahead-right of box ”; nattext b; “? ”.
    keys? screen!.  x:= textnat keys.
    box:= (b; “ahead right”) → x | (x; “behind right”) → b | box.
    screen! “What box is beside box ”; nattext b; “? ”.  keys? screen!.
    box:= (b; “beside”) → textnat keys | box.
    screen! “What are the x and y coordinates of box ”; nattext b; “? ”.
    keys? screen!.  box:= (b; “x”) → textnat keys | box.
    keys? screen!.  box:= (b; “y”) → textnat keys | box.
    draw b “grey” od.` default; may be changed below

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

init_randomnat 123456789. ` initialize a random number generator

` end of initialization, start of simulation

infiniteloop do time? real.  new iterationstarttime:= time.

                new p: (index “porter”) = 0. ` p:= the porter that arrived at its current position first
                new t: (real×s) = 10↑38×s.` t is a time, and 10↑38 is an approximation to ∞
                for q:= index “porter”
                do if porter q “arrival time” < t then t:= porter q “arrival time”.  p:= q else ok 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 ok
                else 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
new b: (index "box") = (boxesToDo↓0) 0. ` the box we are looking at
new d: (nat×m) = (boxesToDo↓0) 1. ` its distance ahead of porter p
new calculateAccel ` of porter p due to porter pa if any
do ⟨ pa: (index "porter", numporters) →
      if pa=numporters then ok
      else new desiredspeed:=
                 (    sqrt (porter pa "speed"↑2 + 2×maxaccel×d + (maxaccel×cushion)↑2)
                   – maxaccel×cushion   ) ∧ speedlimit.
            accel:= ((desiredspeed–porter p "speed")×impatience ∨ –maxaccel) ∧ accel
            fi ⟩ od.

nextbox do  b:= (boxesToDo↓0) 0.  d:= (boxesToDo↓0) 1.
                boxesToDo:= boxesToDo↓(1;..↔boxesToDo).
                if d>maxdistance then ok
                else calculateAccel (box b "above").
                      calculateAccel (porter (box b "beside") "above").
                      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 ok
                             else boxesToDo:= boxesToDo; [box b "ahead right"; d+1×m] fi.
                             nextbox
                      else if ↔boxesToDo > 0 then nextbox else ok fi fi fi od.
old b. old d. old calculateAccel. 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".
next_randomnat.
b:= box b if value_randomnat 0 2 = 0 then "ahead left" else "ahead right" fi.
if box b "porter" = numporters then ok else 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.

await (iterationstarttime+visualdelaytime).
old speed. old accel. old p. old iterationstarttime.
infiniteloop od od
```

Grammar LL($^1/_2$)

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 why I call it LL($^1/_2$). The parse stack begins with only the program nonterminal on it, and ends empty with no more input.

program	process programafterprocess	
process	phrase processafterphrase	
programafterprocess	‖ process programafterprocess	
	empty	
phrase	**new** newname phraseafternewname	
	old oldname	
	open dictionaryname	
	do program **od** arguments	
	if data **then** program **else** program **fi** arguments	
	for simplename := data **do** program **od**	
	⟨ simplename parameterkind primary → program ⟩ arguments	
	indvarname := data	
	channelname afterchannelname	
	newname **do** program **od**	
	programname arguments	
parameterkind	:	
	::	
	!	
	?	
afterchannelname	! data	
	? data echo	if echo is ! then data must be channelname
echo	!	
	empty	
processafterphrase	. phrase processafterphrase	
	empty	
phraseafternewname	: primary = data	
	= data	
	:= data	
	! ? data	
	do program **od**	
	open	
	empty	
data	comparand aftercomparand	
comparand	element afterelement	
element	item afteritem	
item	term afterterm	
term	factor afterfactor	
factor	# factor	
	– factor	
	~ factor	
	+ factor	
	? factor	
	□ factor	
	ϟ factor	

	* factor
	primary factorafterprimary
primary	number
	text
	if data **then** data **else** data **fi** arguments
	result simplename : primary = data **do** program **od** arguments
	{ data }
	[data] arguments
	(data) arguments
	⟨ simplename : primary → data ⟩ arguments
	indvarname arguments
	depvarname arguments
	constantname arguments
	channelname
arguments	number arguments
	text arguments
	if data **then** data **else** data **fi** arguments
	result simplename : data **do** program **od** arguments
	{ data } arguments
	[data] arguments
	(data) arguments
	⟨ simplename : primary → data ⟩ arguments
	indvarname arguments
	depvarname arguments
	constantname arguments
	channelname arguments
	empty
aftercomparand	= comparand aftercomparand
	< comparand 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
afteritem	; item afteritem
	;.. item afteritem
	‘ item afteritem
	empty
afterterm	+ term afterterm
	– term afterterm
	$^+$ term afterterm
	∪ term afterterm

	empty
afterfactor	× factor afterfactor
	/ factor afterfactor
	∩ factor afterfactor
	∧ factor afterfactor
	∨ factor afterfactor
	Δ factor afterfactor
	∇ factor afterfactor
	@ factor afterfactor
	empty
factorafterprimary	↑ factor
	↓ factor
	→ factor
	* factor
	empty
name	simplename compounder
compounder	_ dictionaryname compounder
	empty
newname	simplename not previously defined in the current scope
oldname	simplename previously defined in the current scope
indvarname	name defined as independent variable or variable parameter or **result** variable
depvarname	name defined as dependent variable
constantname	name defined as constant or constant parameter or **for** parameter or **unit**
channelname	name defined as a channel
programname	name defined as a program or procedure
dictionaryname	name defined as a dictionary

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, so they can be eliminated: program process name data comparand element item term. The nonterminal name is used only in the informal productions at the end.

Grammar LR($^1/_2$)

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 is essentially “looking at” an input symbol. So I'll compromise and call it LR($^1/_2$). The parse stack begins empty, and ends with only the program nonterminal on it and no more input.

program	process
	program ‖ process
process	phrase
	process . phrase
phrase	**new** newname : primary = data
	new newname = data
	new newname := data
	new newname **do** program **od**
	new newname ! ? data
	new newname **open**
	new newname **unit**

	new newname	
	old oldname	
	open dictionaryname	
	close dictionaryname	
	indvarname := data	
	channelname ! data	
	channelname ? data	
	channelname ? data !	data must be channelname
	newname **do** program **od**	
	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	
	programname	
data	data = comparand	
	data ≠ comparand	
	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 ; item	
	element ;.. item	
	element ‘ item	
	item	
item	item + term	
	item – term	
	item $^{+}$ term	
	item ∪ term	
	term	
term	term × factor	
	term / factor	
	term ∧ factor	
	term ∨ factor	
	term Δ factor	
	term ∇ factor	
	term ∩ factor	

	factor
factor	+ factor
	– factor
	# factor
	~ factor
	? factor
	□ factor
	ϟ factor
	* factor
	primary * factor
	primary → factor
	primary ↑ factor
	primary ↓ factor
	primary
primary	primary argument
	primary @ argument
	argument
argument	number
	text
	[data]
	{ data }
	(data)
	⟨ simplename : primary → data ⟩
	if data **then** data **else** data **fi**
	result simplename : primary = data **do** program **od**
	indvarname
	depvarname
	constantname
	channelname
name	simplename
	name _ simplename
newname	simplename not previously defined in the current scope
oldname	simplename previously defined in the current scope
indvarname	name defined as independent variable or variable parameter or **result** variable
depvarname	name defined as dependent variable
constantname	name defined as constant or constant parameter or **for** parameter or **unit**
channelname	name defined as a channel
programname	name defined as a program or procedure
dictionaryname	name defined as a dictionary

The nonterminal name is used only in the informal productions at the end.