ProTem

Eric Hehner

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

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

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

```plaintext
« n≥0 ⇒ n’=0 » do if n>0 then n:= n–1. « n≥0 ⇒ n’=0 » fi od
```

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

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

A primary design criterion is to make ProTem a small, easy-to-learn, easy-to-use language. The size of a language can be measured by the number of symbols and amount of grammar structure. ProTem is presented by a Presentation Grammar, which has just the structure that a programmer needs to know, not all the structure that a parser needs for parsing. There is also an LL(1) grammar and an LR(0) grammar; they are at the end of this document. But we begin the document with the Presentation Grammar. It has 2 nonterminals (program and data) plus some informally defined kinds of names. (The LL(1) grammar has 24 nonterminals; the LR(0) grammar has 12 nonterminals. For comparison, the Haskell grammar has 68 nonterminals, and the Python grammar has 87 nonterminals.) ProTem has 12 keywords. (C has 28, Python has 33, Pascal has 36, Haskell has 37, Ada has 62, MS Basic has 205.) The design ethos was that adding a new feature to ProTem requiring a new keyword requires an extremely good reason. That same design ethos will not tolerate any addition to the 2 nonterminals in the Presentation Grammar.

To judge ease of use, one needs to use the language, but one may get a sense of the ease of use from reading example programs. (One may also get a sense of the beauty of the language from example programs, if that's of interest.) For that purpose, there are example programs near the end of this document.

The language design is complete except for the following. We need to describe and compose graphical elements. We need to define touchpad and touchscreen gestures. We need a sound (noise) data type. We need to define regions of documents and regions of the screen to be clickable links.
Symbols

ProTem uses letters, digits, and a blank space. In addition, there are 12 keywords, plus 4 kinds of lexeme, and 63 other symbols; altogether they are:

```
if  then  else  fi  new  old  for  do  od  result  dictionary  unit
number  text  name  comment
```

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

```
for " use " for " use " for « use << for » use >>
for | use |= for ≤ use <= for ≥ use >= for ' use '
for 〈 use 〈 for 〉 use 〉
for – use - for × use &
for ↑ use ^ for ↓ use \ for –> use <>
for ∧ use / \ for ∨ use / for < > use "
for ∈ use elt for ⊆ use sub for ∪ use cup
for ⊙ use cap for ⊗ use [ ] for ∆ 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. Here are four examples.

```
0     275     27.5     0.21
```

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. Characters within a text are not limited to any alphabet. Here are five examples.

```
""          "abc"         "don't"         "Just say "no"").         "♠♣♥♦"
```

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

```
plain simple names:  x  AI  george  refStack
fancy simple names:  «William & Mary»  «x' ≥ x»
compound names:  ProTem_grammars_LL1  DCS_«grad recruiting»_«2016-9-8»
```

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

Presentation Grammar

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

- a newname: a simple name that is not defined in the current scope,
- or a compound name that is not defined in its dictionary
- an oldname: a simple name that is defined in the current scope,
- or a compound name that is defined in its dictionary
At each point in a program, an oldname is one of:
- variablename: a name defined as a variable
- constantname: a name defined as a constant, including unit names
- dataname: a name defined as data
- programname: a name defined as a program or procedure
- channelname: a name defined as a channel
- dictionaryname: a name defined as a dictionary

There are 57 ways of expressing data. Some examples, explanations, and pronunciations are shown.

number

data %   0   1.2
+ data   percentage, divide by 100
− data   plus, identity
data + 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, exclusive or
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
c data  bunch size
\{ data \} set
\~ data  contents of a set or list
data ∈ data elements of a set
data ⊆ data subset
data ∪ data set union
data ∩ data set intersection
\# data  power
\$ data  set size
\text data  “abc”
data ; data string catenation
data ... data string from(including) to(excluding)
data ↓ data string indexing
data < data string modification
data >= data string length
data * data definite repetition
* data indefinite repetition
[ data ]
# data
data † data
data data
data @ data
〈 simplename : data → data 〉
data → data
□ data
data | data
variablename
constantname
dataname
channelname
? channelname
if data then data else data fi
result simplename : data := data do program od
( data )

There are 29 ways of forming a program.

new newname : data := data
c new newname := data
c new newname = data
c new newname do program od
c new newname ? ! data
c new newname unit
c new newname dictionary
c new newname oldname
c new newname
old oldname
variablename := data
channelname ! data
channelname ? data
channelname ? ! channelname
newname do program od
programname
〈 simplename : data → program 〉
〈 simplename :: data → program 〉
〈 simplename ! data → program 〉
〈 simplename ? data → program 〉
program data
program variablename
program channelname
program . program
program || program
if data then program fi
if data then program else program fi
for simplename := data do program od
do program od
Here is the precedence of the forms of program.

0. \texttt{if then fi} \hspace{1em} \texttt{if then else fi} \hspace{1em} \texttt{for do od} \hspace{1em} \texttt{do od} \hspace{1em} \langle \rangle \hspace{1em} \texttt{programname}

1. program argument
2. := ! ?
3. \|
4. .

Program parentheses \texttt{do od} can always be used to group programs differently.

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

0. number text name ( ) [ ] \{ \} \langle \rangle \hspace{1em} \texttt{if then else fi} \hspace{1em} \texttt{result do od}

1. juxtaposition \hspace{1em} \% \hspace{1em} @ \hspace{1em} left-to-right

2. + \hspace{1em} – \hspace{1em} ¢ \hspace{1em} $ \hspace{1em} ↔ \hspace{1em} \# \hspace{1em} ~ \hspace{1em} ? \hspace{1em} ☐ \hspace{1em} * \hspace{1em} \rightarrow \hspace{1em} \uparrow \hspace{1em} \downarrow \hspace{1em} \hspace{1em} \text{prefix} \hspace{1em} + \hspace{1em} – \hspace{1em} ¢ \hspace{1em} $ \hspace{1em} ↔ \hspace{1em} \# \hspace{1em} ~ \hspace{1em} ? \hspace{1em} ☐ \hspace{1em} * \hspace{1em} \rightarrow \hspace{1em} \uparrow \hspace{1em} \downarrow \hspace{1em} \text{right-to-left}

3. \times \hspace{1em} / \hspace{1em} \cap \hspace{1em} \wedge \hspace{1em} \vee \hspace{1em} \Delta \hspace{1em} \nabla \hspace{1em} \text{infix} \hspace{1em} \text{left-to-right}

4. + \hspace{1em} – \hspace{1em} † \hspace{1em} \cup \hspace{1em} \text{infix} \hspace{1em} – \hspace{1em} \text{left-to-right}

5. ; ;.. ; \hspace{1em} \| \hspace{1em} \|.. \| \hspace{1em} \| \hspace{1em} \|.. \| \hspace{1em} \| \hspace{1em} \text{infix} \hspace{1em} \text{left-to-right}

6. = \hspace{1em} \neq \hspace{1em} \leq \hspace{1em} \geq \hspace{1em} : \hspace{1em} \subseteq \hspace{1em} \text{infix continuing}

On level 7, the operators are “continuing”. This means, for example, that \( a=b=c \) neither associates to the left \((a=b)=c\) nor associates to the right \(a=(b=c)\), but means \((a=b)\land(b=c)\). Similarly \(a<b=c\) means \((a<b)\land(b=c)\), and so on.

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

\textbf{Data}

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

\textbf{Numbers}

Numbers are not divided into disjoint types. A natural number is an integer number; an integer number is a rational number; a rational number is real number; a real number is a complex number.

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. The postfix operator \% means division by 100; for examples, 99\%, x\% and \((x+y)\%\). There are 1-operand prefix operators + and –. There are 2-operand infix operators \(+\hspace{1em}–\hspace{1em}\times\hspace{1em}/\hspace{1em}\uparrow\). There are predefined function names such as \(abs, exp, log, ln, sin, cos, tan, ceil, floor, round, re, im, sqrt, div, and mod\) (see \textbf{Predefined Names}). Division of integers, such as \(1/2\), may produce a noninteger. Exponentiation is 2-operand infix \(\uparrow\); for example, \(1.2\times10\uparrow3\) (one point two times ten to the power three). The operator \(\wedge\) is minimum (arms down, does not hold water). The operator \(\vee\) is maximum (arms up, holds water). The operator \(\Delta\) is the negation of minimum. The operator \(\nabla\) is the negation of maximum.
Characters

A character is a text of length 1. We leave it to each implementation to list the characters, and to state their order. In addition to the character symbols such as “a” (small a) and “ ” (space), there are six predefined character names: backspace, tab, newline, click, doubleclick, and end (the end-of-file character). Predefined functions 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.

Bunches

There are several predefined bunch names:

null empty
nat all natural numbers. Examples: 0, 1, 2
int all integer numbers. Examples: –2, –1, 0, 1, 2
rat all rational numbers. Example: 1/2
real all real numbers. Example: 2↑(1/2)
com all complex numbers. Example: (–1)↑(1/2)
char all characters. Example: “a”
bin both binary values: true, false
text all texts (character strings). Example: “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≤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 ψ. For examples, ψ(0, 1) = 2 and ψ(null) = 0 and ψ(a,..,b) = b–a.
Bunches are equal if and only if they consist of the same elements, ignoring order and multiplicity.

In ProTem, all operators whose precedence is before that of bunch union, except \( \cup \), 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 \((\text{nat}+2)\), the even naturals \((\text{nat}\times2)\), the square naturals \((\text{nat}^\uparrow 2)\), the natural powers of two \((2^\uparrow\text{nat})\), 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\}, \{\text{nat}\}. The inverse of set formation is \(~\). For example, \(~\{0, 1\} = 0, 1\). The size of a set is \(\$\). For examples, \$\{0, 1\} = 2 and \$\{null\} = 0. The element \(\in\), subset \(\subseteq\), union \(\cup\), and intersection \(\cap\) operators are as usual. The power operator \(\times\) takes a bunch as operand and produces all sets that contain only elements of the bunch. For example, \(\times\{0, 1\} = \{\text{null}\}, \{0\}, \{1\}, \{0, 1\}\).

Strings

There is a predefined string name:
\[\text{nil}\] 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 \(x \leq y\). Again, \(x\) is included and \(y\) is excluded. For examples, \(0..10\) is a string consisting of the first ten natural numbers, and \(5..5 = \text{nil}\).

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

A string is indexed by the \(\downarrow\) operator. Indexing is from 0. For example, \((2; 3; 5; 7)\downarrow2 = 5\). A string can be indexed by a string. For example, \((3; 5; 7; 9)\downarrow(2; 1; 2) = 7; 5; 7\).

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

A text is a more convenient notation for a string of characters.
\[
\text{“abc”} = \text{“a”}; \text{“b”}; \text{“c”}
\]
\[
\text{“He said “Hi”“} = \text{“H”}; \text{“e”}; \text{“”}; \text{“s”}; \text{“a”}; \text{“i”}; \text{“d”}; \text{“”}; \text{“”}; \text{“H”}; \text{“i”}; \text{“”}; \text{“”}; \text{“”}
\]
\[
\text{“abcdefghij”} \downarrow (3;..6) = \text{“def”}
\]

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

\[ n \ast S \]

is a string, and

\[ \ast S \]

is a bunch of strings. For examples,

\[ 3 \ast 5 = 5;5;5 \]
\[ 3 \ast (4, 5) = 4;4;4, 4;4;5, 4;5;4, 4;5;5, 5;4;4, 5;4;5, 5;5;4, 5;5;5 \]
\[ \ast 5 = nil, 5, 5;5, 5;5;5, \text{ and so on} \]

The \( \ast \) operator distributes over bunch union, but in its left operand only.

\[ \text{null} \ast 5 = \text{null} \]
\[ (2,3) \ast 5 = (2 \ast 5),(3 \ast 5) = 5;5, 5;5;5 \]

Using this semi-distributivity, we have

\[ *a = \text{nat} \ast a \]

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 \( L \)
- \~L: content of \( L \)
- \( L n \): \( L \) at index \( n \)
- \( L @ p \): \( L \) at pointer \( p \)
- \( L \uparrow M \): \( L \) catenate \( M \)
- \( L M \): \( L \) composed with \( M \)
- \( L \mid M \): \( L \) otherwise \( M \), the selective union of \( L \) and \( M \)
- \( i \rightarrow x \mid L \): index \( i \) is item \( x \) and otherwise \( L \)

plus the comparisons \( L=M \), \( L\uparrow M \), \( L<M \), \( L>M \), \( L\leq M \), \( L\geq M \).

Here are some examples.

\[ \#[0; 1; 2] = 3 \]
\[ \sim[0; 1; 2] = 0;1;2 \]
\[ [0;\ldots;10] 5 = 5 \]
\[ [2; 3]; 4; [5; [6; 7]]] @ (2; 1; 0) = 6 \]
\[ [0;\ldots;10] \uparrow [10;\ldots;20] = [0;\ldots;20] \]
\[ [10;\ldots;20] [3; 6; 5] = [13; 16; 15] \]

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 \( L \uparrow M \) is indexed with \( n \), the result is either \( L n \) or \( M n \) depending on whether \( n \) is in the domain \((0,\ldots;\#L)\) of \( L \). If it is, the result is \( L n \), otherwise the result is \( M n \).

\[ [10; 11] \uparrow [0;\ldots;10] = [10; 11; 2;\ldots;10] \]
\[ 1 \rightarrow 21 \mid [10; 11; 12] = [10; 21; 12] \]
The index can be a string, as in

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

When a string or list is indexed by a structure, the result has that same structure. For example, let $S = 10; 11; 12$. Then

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

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

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

Lists are equal if and only if they 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 $[10*\text{nat}]$ is all lists of length 10 whose items are natural, and $[4*[6*\text{real}]]$ is all 4 by 6 arrays of reals.

Conditional Data

The 3-operand $\text{if } x \text{ then } y \text{ else } z \text{ fi}$ has binary operand $x$, but $y$ and $z$ are of arbitrary type. For example,

$$\text{if } \text{reply} = \text{"yes"} \text{ then } 0 \text{ else } \text{"nan"} \text{ fi}$$

If $\text{reply} = \text{"yes"}$ has value $\text{true}$, then this data expression has number value $0$. If it has value $\text{false}$, then this data expression has text value “nan”.

Functions

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

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

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

$$\langle n; \text{nat} \rightarrow n+1 \rangle \quad \text{map } n \text{ in } \text{nat} \text{ to } n+1$$

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

$$\langle a; \text{real} \rightarrow \langle b; \text{real} \rightarrow \text{if } a > b \text{ then } a \text{ else } b \text{ fi} \rangle \rangle$$

Similarly for functions with more than two parameters.

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

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,

\[
2 \rightarrow 3
\]

abbreviates \( \langle n: 2 \rightarrow 3 \rangle \) or choose any other parameter name.

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

\[
\langle n: nat \rightarrow mod n 2 = 0 \rangle : nat \rightarrow bin
\]

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

\[
(f, g)(x, y) = fx, fy, gx, gy
\]

Programmed Data

Programmed data allows us to use a program to compute data.

result simplenname : data := data do program od

First, a local variable is defined with a type and initial value; its scope (see Scope, next) is from do to od. Then the program is executed. The result is the final value of the newly defined local variable. We have not yet presented programs, but the following example, which approximates the base of the natural logarithms \( e \), should give the idea.

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

There are no side effects. Nonlocal variables become constants within the result expression; their values may be used, but assigning them is not permitted. Input and output are not permitted.

All the ways of expressing data can be combined arbitrarily, without restriction. Here is a function whose body is programmed data. It expresses the number of times 2 is a factor of \( n \).

\[
\langle n: (nat+1) \rightarrow result f: 0,..n:= 0 do new m: 1,..n+1:= n. loop do if mod m 2 = 0 then f := f+1. m:= m/2. loop fi od od\rangle
\]

A result variable begins its scope after the corresponding do and ends its scope at the corresponding od. Consequently, the result variable can be any simple name, even one that has already been defined in the scope that encloses the programmed data.

Scope

A name is defined in these ways: by the keyword new, as a named program, as a parameter just after \( \langle \rangle \), as a for-index, or as a result variable. We shall come to each of these shortly. The scope of a name is the part of a program in which the name is defined. We shall also come to the ways of composing larger programs from smaller programs using program brackets do od, and conditional programs if then fi and if then else fi. And we can parameterize data and programs using angle brackets \( \langle \rangle \). Scopes are limited by do od, then fi, then else, else fi, and \( \langle \rangle \). Each of these five pairs is a scope opener and a scope closer.
A simple name defined using the keyword `new` must be new, not already defined, since the most recent scope opener. Its scope extends from its definition, through all following sequentially composed programs, to the corresponding scope closer. But it may be covered by a redefinition in an inner scope. Using `new x = 2` and `new x = 3` as example definitions, and the program brackets `do od` as example scope limiters, and letting `A`, `B`, `C`, `D`, and `E` stand for arbitrary program forms (but not `new` or `old`), in

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

the definition of `x` as the number 2 is not yet in effect in `A`, but it is in effect in `B`, `C`, and `E`. The definition that makes `x` the number 3 is in effect in `D`. None of `A`, `B`, `C`, `D`, or `E` can contain a redefinition of `x` unless it is within further scope limiters `do od`, `then fi`, `then else`, `else fi`, or `{ }`.

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

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

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

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

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

A scope can be nested inside another scope, which can be nested inside another, and so on. If a name is defined 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 defined outside all scope limiters serve as “files”.

Outside the outermost scope that you can use, there is a superscope where the predefined names are defined. They are usable in all your scopes unless you cover them by redefining the names. You cannot end the scope of a predefined name.

**Programs**

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

**Variable Definition**

Here is an example variable definition.

```
  new x: nat:= 5
```

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

```
  new y: 0..2*x:= x
```

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

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

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

**Assignment**

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

```
x:= x+1
s:= 3 among s
```

The data on the right of `:=` must be an element in the type of the variable on the left of `:=`. As in the examples, the data on the right of `:=` can make use of the variable on the left of `:=`.

**Constant Definition**

Here are three constant definitions.

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

where \( pi \) is a predefined constant name.

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

```
new xplus1:= x+1
```

The current value of variable \( x \) is used to evaluate \( x+1 \), and \( xplus1 \) expresses that value. Variable \( x \) may later be reassigned to another value, but that does not affect the value of \( xplus1 \). Constant name \( xplus1 \) cannot be reassigned.

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

**Data Definition**

The data definition

```
new xplus2 = x+2
```

makes the value of \( xplus2 \) depend on the value of variable \( x \). As \( x \) changes value, \( xplus2 \) changes value so that \( xplus2 = x+2 \) is always true. In the constant definition of \( xplus1 \) earlier, \( x+1 \) is evaluated once, at definition time. By contrast, in the data definition of \( xplus2 \), \( x+2 \) is not evaluated at definition time; it is evaluated every time \( xplus2 \) is used.

A data definition can depend indirectly on a variable. For example,

```
new twoxplus4 = 2*xplus2
```

makes \( twoxplus4 \) depend indirectly on the value of variable \( x \).
Data Recursion

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

```
new no: 0..2×no:= no  ` illegal
```

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

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

```
new fact = 0 → 1 | n: (nat+1) → n × fact (n–1)

new div = 〈 a: nat → 〈 d: (nat+1) →
               if a<d then 0 else if even a then 2 × div (a/2) d else 1 + div (a–d) d fi fi) 〉
```

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

```
new exp =  “a”, “b”, exp; “+”; exp, “–”; exp
```

This recursive definition is equivalent to the nonrecursive definition

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

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

```
new eat = 〈 a: nat → if a=0 then 0 else eat fi)
```

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

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]
```

Constant Definition versus Data Definition

As already stated, a constant definition evaluates its data once, at definition time, whereas a data definition evaluates its data at each use. If the data is fully evaluated, there is no difference. For example, there is no difference between

```
new five:= 5
new five = 5
```

When there are no variables used to express the value (neither directly nor indirectly), there is no semantic difference between data definition and constant definition, but there may be an efficiency difference. Here is a trivial example.

```
new csix = 5+1
new dsix = 5+1
```

If the definition is never used, `dsix` is more efficient. If the definition is used once, they are equally efficient. If the definition is used two or more times, `csix` is more efficient. Here is a more interesting example.

```
new cdouble:= 〈 n: (0..10) → 2×n)  
new ddouble = 〈 n: (0..10) → 2×n)  
```

The constant definition `cdouble` causes the function to be evaluated. That means that the function is applied to all its arguments, and all the results are stored. In effect, the function is evaluated to the
When *cdouble* is used by applying it to an argument, that argument indexes the list. The data definition *ddouble* does not evaluate the function. Each time *ddouble* is used by applying it to an argument, the body of the function is evaluated. Which one is more efficient depends on the size of the domain, the complexity of the result, and the number of times the definition is used.

**Program Definition**

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

```plaintext
new switchends do s:= 0 \rightarrow s 919 \rightarrow s 01 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.

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

```plaintext
new km := 1000\times m
```

to make *km* be a kilometer, and

```plaintext
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,

```plaintext
new speed: real\(\times m/s := 3.6\times km/h
```

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

```plaintext
new q: 25\times sheet. new ream := 20\times quire.
new order: nat\times sheet := 3\times ream
```

assigns *order* to 1500\(\times sheet\). When the value 5\(\times m/s\) is converted to text by *numtext*, the result is “5 m/s” without the \(\times\) sign and without evaluating the unknown real value *m/s*.

**Dictionary Definition**

Dictionaries are the way you organize your data. You can create as many dictionaries as you want. To create a new dictionary named *abc*, write

```plaintext
new abc dictionary
```

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

```plaintext
new abc_x := 2
```

defines *x* in dictionary *abc* to be the data 2. A name can be removed from a dictionary using the keyword *old*. For example

```plaintext
old abc_x
```

When a dictionary name is removed, the dictionary remains in existence, anonymously, as long as something refers to it or to its contents.
In the superscope where the predefined names are defined, there is also a dictionary named `predefined` with all the predefined names and definitions in it. So if you cover a predefined name, say `i`, with a redefinition, you can still refer to the predefined `i` as `predefined_i`.

When a name in a dictionary is defined to be a dictionary, this dictionary also contains names, some of which can be defined as dictionaries, and so on. So a dictionary can be a tree structure. Suppose there is a dictionary named `ProTem` within which there is a dictionary named `grammars` within which there is a text named `LL1`. Its name is `ProTem_grammars_LL1`.

**Synonym Definition**

The definition

```
new LL1  ProTem_grammars_LL1
```

makes `LL1` a synonym for `ProTem_grammars_LL1`. One use, as in the example, is to shorten a name.

If a predefined name is covered by a definition in a scope between the superscope where it is defined and the local scope where you are working, it can be recovered using a synonym. For example, if predefined `i` has been redefined, it can be recovered by

```
new i predefined_i
```

Then any use of `i` is a use of `predefined_i`, except that `old i` removes the local name `i`, not the name `predefined_i`.

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

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

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

**Name Removal**

Names defined with the keyword `new` can be undefined with the keyword `old`. Ironically, by saying `old x`, the name `x` becomes available for reuse as a new name. Even though a name may be undefined, 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 and ready for use. The name `s` was defined for the purpose of defining the other names, and then removed, leaving the other names dependent upon an anonymous variable.
The predefined names include \texttt{randomNat}, \texttt{randomNatInit}, and \texttt{randomNatNext}. They might have been defined as:
\begin{verbatim}
new \texttt{big}:= 2^{31}.
new \texttt{rv}: 0..\texttt{big}:= 123456789.
new \texttt{randomNat} = \{\texttt{from}: \texttt{nat} \rightarrow \{\texttt{to}: \texttt{nat} \rightarrow \texttt{floor (from + (to–from)\times rv/bigr)}\}\}.
new \texttt{randomNatInit} do \{\texttt{seed}: (0..\texttt{big}) \rightarrow \texttt{rv}:= \texttt{seed}\} od.
new \texttt{randomNatNext} do \texttt{rv}:= \text{mod (rv \times 5^{13}) \texttt{big}} od.
\end{verbatim}
old \texttt{big}. old \texttt{rv}

Constant \texttt{big} and variable \texttt{rv} are now hidden; their names are removed, but \texttt{randomNat}, \texttt{randomNatInit}, and \texttt{randomNatNext} still use them. We can use these definitions as follows:
\begin{verbatim}
randomNatInit 555555555.
randomNatNext.
screen! numtext (randomNat 0 10)
\end{verbatim}

Sequential Composition

Sequential composition is denoted by a period. It is an infix connective. In other words, the period comes between and joins two programs. Unlike the period in English, the period in ProTem does not come at the end of a program that is not sequentially followed by another program.

Parallel Composition

The parallel composition of programs $P$, $Q$, and $R$ is $P\|Q\|R$. A variable defined before the parallel composition remains a variable in at most one of the programs in the parallel composition; in all the other programs, it becomes a constant. For example,
\begin{verbatim}
new \texttt{a}: \texttt{nat}:= 1.
new \texttt{b}: \texttt{nat}:= 2.
new \texttt{c} = \texttt{a}+\texttt{b}.
do \texttt{a}:= 4. A \texttt{od} \parallel \texttt{do} \texttt{b}:= 8. B \texttt{od}.
C
\end{verbatim}

In the parallel composition, variable $a$ can be reassigned in one of the parallel programs, but not in both; it is reassigned in the left program. Likewise variable $b$ can be reassigned in one of the parallel programs, but not in both; it is reassigned in the right program. At the start of $A$, variable $a$ has value 4, constant $b$ has value 2, and data $c$ has value 6. At the start of $B$, constant $a$ has value 1, variable $b$ has value 8, and data $c$ has value 9. At the start of $C$, variable $a$ has value 4, variable $b$ has value 8, and data $c$ has value 12. Parallel programs cannot affect each other through assignments of variables. For co-operation, programs can communicate with each other on channels defined for the purpose (see Channel Definition).

Here is a program to find the maximum value in nonempty list $L$ in $\log (\#L)$ time. ($L$ is a variable, and its initial value is destroyed in the process.) We define \texttt{findmax $i$ $j$} to find the maximum in the segment of $L$ from index $i$ to index $j$, reporting the result as $L i$.
\begin{verbatim}
new \texttt{findmax} do \{\texttt{i}: (0..\#L) \rightarrow \{\texttt{j}: (1..\#L+1) \rightarrow \texttt{if} j–i\geq 2 \texttt{then} \texttt{findmax} (i+(i+j) 2) \parallel \texttt{findmax} (div (i+j) 2) j. \texttt{L}:= i \rightarrow (L i \vee (L (div (i+j) 2) \mid L \texttt{fi})\} od
\end{verbatim}
After execution of \texttt{findmax 0 (#L)}, the maximum value in the initial 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. Output is buffered so it will be available when `screen` is ready to receive it. A string of outputs can be sent together

```
screen! "Answer = "; numtext x; newline
```

where `numtext` is a predefined function that converts from a 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. The shift-A combination is a single character “A”. Likewise the control-Q combination is a single character. The click button is just a key like any other: `click` is a character, and `doubleclick` is a character. And `backspace`, `tab` and `newline` 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 a text that can be interpreted as a signed number, possibly preceded by spaces, define

```
new digit:= "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
```

and then input

```
keys? *" "; (+", "+", ";", ";", ";", ";"); digit; *digit; ((". "; digit; *digit); ";")
```

This grammar is predefined, and named `formnum`.

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

```
x:= textnum keys
```

where `textnum` is a predefined function that converts from a text to a number. We may choose to echo the previous input to the screen by saying

```
screen! keys
```

There is a second form of input that reads from a text channel and simultaneously writes on a text channel. For example,

```
keys? !screen
```

reads text from channel `keys`, corrected according to `backspace` characters, up to and including the next `newline` character, and echoes the input on the screen character by character and correction by correction. 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
\[ \text{new } c\text{?! nat} \]
defines \( c \) to be a new local channel that transmits naturals. It can be used for output and input. For example,
\[ \text{new } c\text{?! nat. } c\text{! 7 } \parallel \text{ do } c\text{? 0..100. } x := c \text{ od. } \text{old } c \]
assigns \( x \) to \( 7 \).

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

Conditional Program

The conditional program \( \text{if } a \text{ then } b \text{ fi} \) is executed as follows. Binary expression \( a \) is evaluated; if its value is \textit{true}, then \( b \) is executed; if the value of \( a \) is \textit{false}, then the conditional program has no effect. The conditional program \( \text{if } a \text{ then } b \text{ else } c \text{ fi} \) is executed as follows. Binary expression \( a \) is evaluated; if its value is \textit{true}, then \( b \) is executed; if the value of \( a \) is \textit{false}, then \( c \) is executed. An “assert” program is obtained according to the following example.
\[ \text{if } x>y \text{ then screen! “appropriate error message”. stop fi} \]

Named Programs

A named program has the syntax
\[ \text{name} \text{name} \text{do program od} \]
The name of a named program must be new, just as if it were defined with the keyword \textit{new}. But its scope is just within the \textit{do od} pair that it names. After that, it is again new and can be reused. The name is attached to the program (like a program definition), and the program is executed (unlike a program definition). One purpose of this naming is to make loops. Here is a two-dimensional search for \( x \) in an \( n \times m \) array \( A \) of integers (that is, \( A: [n*[m*int]] \)).
\[ \text{new } i: \text{nat}=0. \]
\[ \text{tryThisI do if } i=n \text{ then screen! numtext } x; \text{“ does not occur.”} \]
\[ \text{else new } j: \text{nat}=0. \]
\[ \text{tryThisJ do if } j=m \text{ then } i:=i+1. \text{tryThisI} \]
\[ \text{else if } A \text{i j} = x \text{ then screen! numtext } x; \text{“ occurs at ”; numtext } i; \text{“ ”; numtext } j \]
\[ \text{else } j:=j+1. \text{ tryThisJ fi fi od fi od} \]

The next example is a fast remainder program, assigning natural variable \( r \) to the remainder when natural \( a \) is divided by positive natural \( d \), using only addition and subtraction.
\[ r:=a. \]
\[ \text{outerloop do if } r\geq d \text{ then new dd: nat}=d. \]
\[ \text{innerloop do } r:=r-dd. \text{ dd}=dd+dd. \]
\[ \text{if } r<dd \text{ then outerloop else innerloop fi od fi od} \]

The use of a program name is semantically a call; it means the same as replacing it with the program it names (including the \textit{do od} brackets). The fast remainder example means the same as
\[ r := a. \]

**outerloop**

*do if* \( r \geq d \)*

**then new** \( dd : nat := d. \)

**innerloop**

*do* \( r := r - dd. \)

*if* \( r < dd \)*

**then do if* \( r \geq d \)*

**then new** \( dd : nat := d. \)

**innerloop**

*do* \( r := r - dd. \)

*if* \( r < dd \)*

*then* *outerloop else innerloop fi od fi od

**else do**

*do* \( r := r - dd. \)

*if* \( r < dd \)*

*then* *outerloop else innerloop 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 \( f_0, f_1, f_2, \) and so on, 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. \( n := x. \) \( x := x + 2 + y + 2. \) \( y := 2 \times n \times y + y + 2 \)

*else* \( n := n/2 - 1. \) Fib. \( n := x. \) \( x := 2 \times n \times x + y + 2. \) \( y := n + 2 + y + 2 + x \) fi fi od

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

\[ \langle x' > x \rangle \]

*do* \( x := x + 1 \) od

The specification on the left \( \langle x' > x \rangle \) 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,

\[ \langle x' = 0 \rangle \]

*do if* \( x + 0 \) *then* \( x := x - 1. \) \( \langle x' = 0 \rangle \) fi od

In the *then*-part, the specification \( \langle x' = 0 \rangle \) means exactly what it says, rather than the program that it names. Thus the use of specifications makes complicated fixed-point semantics unnecessary. If the prover fails to understand the specification, or fails to prove the refinement, it informs the programmer, and treats the specification as just a name.

Suppose a name is defined within a loop. For example, the name \( a \) in

\[ infiniteloop do new a := "a". \] screen! a. \] infiniteloop od

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

\[ infiniteloop do new a := "a". \] screen! a. \] infiniteloop od od

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

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

Indexed Program

This example computes the transitive closure of \( A: [n^[n*bin]] \).

\[
\begin{align*}
\text{for } j &:= 0;..n \\
do \text{ for } i &:= 0;..n \\
d \text{ for } k &:= 0;..n \\
\quad \text{do } A := (i;k) \to (A i k \lor (A i j \land A j k)) \mid A \text{ od od od }
\end{align*}
\]

The assignment can be restated as

\[
\begin{align*}
\text{if } A i j \land A j k \text{ then } A := (i;k) \to \text{true} \mid A \text{ fi }
\end{align*}
\]

if you prefer. The name being defined by \texttt{for} is known only within the loop body, and it is known there as a constant, and so it is not assignable. We call it a \texttt{for}-index. In the example, each index takes values 0, 1, 2, and so on up to and including \( n-1 \), but not including \( n \).

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

\[
\begin{align*}
\text{new } n &:= 1000. \\
\text{new } \texttt{prime}: [n*bin] &:= [2*false; (n-2)*true]. \\
\text{for } i &:= 2;..\text{ceil (sqrt } n) \\
\text{do if } \texttt{prime } i \text{ then } \text{for } j &:= i;..\text{ceil (n/i)} \text{ do } \texttt{prime} := (ixj) \to \text{false} \mid \texttt{prime} \text{ od fi od }
\end{align*}
\]

A \texttt{for}-index is “by initial value”, so

\[
\begin{align*}
\text{for } i &:= x; x \text{ do } x := i+1 \text{ od }
\end{align*}
\]

increases \( x \) by 1, not 2.

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

\[
\begin{align*}
\text{for } i &:= 0,..\#A \text{ do } A := i \to 0 \mid A \text{ od }
\end{align*}
\]

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

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

Procedures

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

\[
\langle y: \texttt{real } \to x := x \times y \rangle
\]

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. The argument provides a value for the parameter. For example,

\[
\langle y: \texttt{real } \to x := x \times y \rangle 3
\]

is the same as

\[
x := x \times 3
\]
A procedure's parameter is known only within the procedure body.

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

\( \{i: \text{int } \rightarrow x := i, y := i\} (x+1) \)

gives both \( x \) and \( y \) a final value one greater than \( x \)’s initial value.

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

\( \{x: \text{int } \rightarrow x := 3\} \)

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

\( \{x: \text{int } \rightarrow x := 3. \ y := 4\} \)

cannot be applied to variable \( y \). The main use for variable parameters is probably to affect many files in the same way; for example, a procedure to sort files.

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

\( \{c! \text{ text } \rightarrow c! \text{ “abc”}\} \)

can be applied to any channel that receives text. A procedure with a channel parameter cannot be applied to a channel appearing in the procedure. This example procedure can be applied to any output channel, even one named \( c \), because the nonlocal channel name \( c \) does not (and cannot) appear in the procedure. Likewise,

\( \{c? \text{ text } \rightarrow c?. \ \text{screen! c}\} \)

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

\( \{c! \text{ text } \rightarrow c! \text{ “abc”}. \ d! \text{ “def”}\} \)

cannot be applied to channel \( d \).

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

\[
\text{new } pps \text{ do } \langle a? \text{ rat } \rightarrow \langle b? \text{ rat } \rightarrow \langle c! \text{ rat } \rightarrow \\
\quad a? \text{ rat} \parallel b? \text{ rat}. \ c! a \times b. \\
\quad \text{new } a0 := a. \ \text{new } b0 := b. \ \text{new } d?! \text{ rat.} \\
\quad pps \ a \ b \ d \\
\quad \parallel \text{do } a? \text{ rat} \parallel b? \text{ rat}. \ c! a0 \times b + a \times b0. \\
\quad \text{loop do } a? \text{ rat} \parallel b? \text{ rat} \parallel d? \text{ rat.} \\
\quad \quad c! a0 \times b + d + a \times b0. \ \text{loop od od})\rangle \text{ od}
\]

Since \( \langle \) opens a new scope, the parameter can be any simple name, even one that has already been defined in the enclosing scope. The corresponding \( \rangle \) closes its scope.

Format

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

\[
\begin{align*}
A. & \quad B & \text{for } x := A \text{ do } B \text{ od} \\
\text{or} & \quad A \quad \text{or} & \text{for } x := A \\
A. & \quad \text{B} & \text{do } B \text{ od}
\end{align*}
\]
A \parallel B

or

A
\parallel B

\begin{verbatim}
if A then B else C fi

or

if A then B
else C fi

or

if A then B
else C fi
\end{verbatim}

\begin{verbatim}
result x: A = B do C od

or

result x: A = B
do C od
\end{verbatim}

\begin{verbatim}
\langle x: A \rightarrow \langle y: B \rightarrow C \rangle \rangle

or

\langle x: A \rightarrow \langle y: B \rightarrow C \rangle \rangle
\end{verbatim}

**Miscellaneous**

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

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

The ProTem equivalent of enumerated type is shown here.

```
new color:= “red”, “green”, “blue”.
new brush: color:= “red"
```

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

```
new person:= “name” \rightarrow text \mid “age” \rightarrow nat.
new p: person:= “name” \rightarrow “Josh” \mid “age” \rightarrow 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” \rightarrow 17 \mid p.
p:= “name” \rightarrow “Amanda” \mid “age” \rightarrow 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 the predefined name \( \text{index} \).

```
new index:= text\rightarrow*nat
```

When applied to a text argument, it yields the result \( *nat \). The use of \( \text{index} \) 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" → end | "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
To move it down to the right, reassign it this way:
$p$ := $p$; 2
Thus $p$ is a string of indexes indicating a subtree $t@p$ of $t$. We can replace this subtree with tree $s$ using the assignment
$t$ := $p$ → $s$ | $t$
We can express the information at the node indicated by $p$ as
$t@p$ 1     or     $t@(p; 1)$
and we can replace the information at this node with the integer 6 using the assignment
$t$ := ($p; 1$) → 6 | $t$

To move up in the tree, we just remove the final item of $p$, and to make that easy, the predefined
new $back$ = $\langle p$: (*nat) → $p\downarrow(0;..≅p−1)\rangle$
allows us to move $p$ up to its parent by writing
$p$ := $back$ $p$

The procedure of some other programming languages is a combination of naming and parameterization. For example,
new $transform$ do $\langle$ magnification: real → $\langle$translation: real → $x$: = magnification$x + translation\rangle$ $\rangle$ od

Here is a procedure with one parameter
new $translate$ do $transform$ 1 od
formed by providing one argument to a two-parameter procedure. To provide an argument for just the second parameter is a little more awkward, but not too bad.
new $magnify$ do $\langle$ magnification: real → $transform$ magnification 0$\rangle$ od

We can now obtain a three-times magnification of $x$ in either of these ways.
$magnify$ 3
$transform$ 3 0

In some other programming languages, the “function” is a combination of naming, parameterizing, and programmed data. For example,
new $fact$ = $\langle n$: nat → result $f$: nat := 1 do for $i$: = 0;..$n$ do $f$: = $f\times(i+1)$ od od\rangle
Exception handling is provided by bunch union or by the $|$ operator. For example,

```prolog
new divide = \langle \text{dividend}: \text{com} \rightarrow \langle \text{divisor}: \text{com} \rightarrow \\
\text{if divisor} = 0 \text{ then } \text{“zero divide” else dividend / divisor fi \rangle \rangle
```

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

```prolog
new weekday = \langle d: (0,..7) \rightarrow 1 \leq d \leq 5 \rangle
```

Then in the expression

```prolog
(weekday \mid all \rightarrow \text{“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.

```prolog
inputchoice do if ?c then c? formnum. P \\
else if ?d then d? formnum. Q \\
else inputchoice fi fi od
```

Unix directories are dictionaries. Unix files are variables. The predefined names `definitions` and `content` give us something like the Unix `ls` command. 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.)

```prolog
new trim do \langle in? text \rightarrow \langle out! text \rightarrow \text{repeatedly read from in, trim off leading and trailing space, output to out, until end is read.} \\
\text{The final end is output} \rangle \rangle od.
```

```prolog
new sort do \langle in? text \rightarrow \langle out! text \rightarrow \text{repeatedly read from in until end is read and output} \\
\text{the sorted texts to out. The final end is output} \rangle \rangle 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

```prolog
new pipe?! text. trim keys pipe. sort pipe screen. old pipe
```

Even better:

```prolog
new pipe?! text. trim keys pipe \| sort pipe screen. old pipe
```

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

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 recognize them and check these comments. It is also possible for an implementation to generate such comments on request. Here is the format.

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

They are transitive through “use” and “call” without requiring the implementation to do a transitive closure (it just checks the comments at the head of the needed data names and 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.

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

**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. Data structures, and the functions and procedures that access and update them, can be defined together in one dictionary. If many objects of the same type are wanted, the type can be defined once and used many times.

**Graphics**

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

\[ \text{new } p: \text{pic} := [x*[y*0]] \]

The name *pic* is defined as \([x*[y*(0..z)]]\) where \(x\) is the number of pixels in the horizontal direction, \(y\) is the number of pixels in the vertical direction, and \(z\) is the number of pixel values. A picture can therefore be expressed in the same way as any other two-dimensional array, and one can refer to the pixel in column 3 and row 4 of picture \(p\) as \(p\ _3\ _4\).

Another predefined name is *movie*, defined as *pic*. The operations on movies are just those of strings, 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, modifying, or deleting any definition (variable name, constant name, data name, program name, channel name, or dictionary name). In the editor, control-e exits the editor, throws away old definitions that have been modified or deleted, along with all definitions that depend on them, and compiles and saves the new definitions.

**Security**

Any dictionary may contain a constant definition of the names *readpassword* and *writepassword*. For examples,

\[
\begin{align*}
\text{new readpassword} & := \text{encode } \text{“Smith”}. \text{ my mother’s maiden name} \\
\text{new writepassword} & := \text{encode } \text{“Fred” } \text{ my father’s middle name}
\end{align*}
\]

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

Session

Sessions are defined for each user of a multiuser computer for security and error recovery. When the computer is turned on, a session begins. When control-q is typed, a session ends and a new one begins. When some idle time passes (how much time is a parameter of the system and may be set to infinity), a session ends and a new one begins. When the computer is turned off, a session ends.

At the start of a session, all passwords are required for dictionaries that include password definitions. A password will not be requested twice within the same session for the same dictionary.

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

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. The undo command (control-u) 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, acting 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 or exit editor
- control-q quit session and start a new session
- control-u undo program
- control-s undo session

Intentionally Omitted Features

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

assertion

\[ \text{assert } x \leq y \quad \text{abbreviates } \text{if } x > y \text{ then screen! “assert failure”. stop fi} \]

list item assignment

\[ A 3 := 5 \quad \text{abbreviates } A := 3 \to 5 \mid A \]
\[ A 3 4 := 5 \quad \text{abbreviates } A := (3; 4) \to 5 \mid A \]

name grouping

\[ \text{new } x, y : \text{int} := 0 \quad \text{abbreviates } \text{new } x : \text{int} := 0, \text{ new } y : \text{int} := 0 \]
\[ \text{old } x, y \quad \text{abbreviates } \text{old } x, \text{ old } y \]
\[ \langle a, b : \text{nat} \to a+b \rangle \quad \text{abbreviates } \langle a : \text{nat} \to \langle b : \text{nat} \to a+b \rangle \rangle \]
\[ \langle a, b : \text{nat} \to x := a+b \rangle \quad \text{abbreviates } \langle a : \text{nat} \to \langle b : \text{nat} \to x := a+b \rangle \rangle \]
\[ x, y := 0 \quad \text{abbreviates } x := 0, y := 0 \]

looping constructs

\[ \text{while } n > 0 \quad \text{loop } n := n-1 \quad \text{pool} \quad \text{abbreviates} \]
\[ \text{loop } n := n-1 \quad \text{do if } n > 0 \text{ then } n := n-1. \text{ loop fi od} \]
\[ \text{loop } n := n-1 \quad \text{until } n = 0 \quad \text{pool} \quad \text{abbreviates} \]
\[ \text{loop } n := n-1. \quad \text{if } \neg(n = 0) \text{ then loop fi od} \]
\[ \text{loop } n := n-1. \quad \text{exit when } n = 0. \quad m := m+1 \quad \text{pool} \quad \text{abbreviates} \]
\[ \text{loop } n := n-1. \quad \text{if } \neg(n = 0) \text{ then } m := m+1. \quad \text{loop fi od} \]

Implementation Philosophy

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

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

\[ \text{screen! numtext } (-3) \quad \text{prints } -3 \]

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

\[ \text{screen! numtext } (1/0) \quad \text{should print } 1/0 \]
\[ \text{screen! numtext } ((0; 1) 2) \quad \text{should print } [0; 1] 2 \]
\[ \text{screen! numtext } ((r : \text{rat} \to 5) (1/0)) \quad \text{should print } 5 \]
\[ \text{screen! bintext } (1/0 = 1/0) \quad \text{should print } \text{true} \]
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 definition, the type must be one of

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

where $n$ is a natural number and $type$ is any of these types just listed. 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. For another example, an implementer may decide not to implement parallel execution.
Predefined Names

Here are the predefined names. Each name is one of:

- **variable** indicated by `var` (evaluated; assignable)
- **constant** indicated by `con` (evaluated; not assignable)
- **data** indicated by `dat` (unevaluated; evaluation upon use; not assignable)
- **program** indicated by `pro` (unexecuted; execution upon use)
- **channel** indicated by `cha`
- **dictionary** indicated by `dic`

Some definitions use `§` or `∃`, which are defined in *a Practical Theory of Programming*.

- `abs`: `com → real` `dat` Absolute value. `abs x = sqrt(re x ↑ 2 + im x ↑ 2)`.
- `all`: `dat` All ProTem items.
- `arc`: `com → § <r: real → 0 ≤ r < 2×π> dat` An approximation to the angle or arc of a complex number.
- `arccos`: `§ <r: real → –1 ≤ r ≤ +1> → § <r: real → 0 < r < pi/2> dat` An approximation to a trigonometric function.
- `arcsin`: `§ <r: real → –1 ≤ r ≤ +1> → § <r: real → 0 < r < pi/2> dat` An approximation to a trigonometric function.
- `arctan`: `real → § <r: real → 0 < r < pi/2> dat` An approximation to a trigonometric function.
- `asm`: `pro` A machine-dependent program with one text input parameter. If the input represents an assembly-language program, the execution is that of the represented assembly-language program.
- `await`: `pro` A program with one constant parameter of type `real×s`. If the argument represents the present or a future time, its execution does nothing but takes time until the instant given by the argument. If the argument represents the present or a past time, its execution does nothing. See `time` and `wait` and `s`.
- `back`: `*nat → *nat` `dat` `back (s; i) = s`.
- `backspace`: `char con` `bin = true, false` `con` `bintext: bin→text` `con` `bintext true = “true”` and `bintext false = “false”`.
- `ceil`: `real→int` `dat` `r ≤ ceil r < r+1`
- `char`: `dat` The characters.
- `charnat`: `char→nat` `dat` A one-to-one function with inverse `natchar`.
- `click`: `char con` `com`: `dat` The complex numbers.
- `content`: `text→text` `dat` A text data name whose value is a readable summary of the content of the named dictionary. For example, `content “abc”`
- `cos`: `real → § <r: real → –1 ≤ r ≤ +1> dat` An approximation to a trigonometric function.
- `cosh`: `com→com` `dat` An approximation to a hyperbolic function.
- `cursor`: `nat; nat` `dat` A data name whose value is the current cursor position.
- `definitions`: `text` `dat` The names and kinds (variable, constant, data, program, channel, dictionary) defined in the current scope.
- `div`: `real → § <r: real → r>0> → int` `dat` `div a d` is the integer quotient when `a` is divided by `d`.
  
  `(0 ≤ mod a d < d) ∧ (a = div a d × d + mod a d)`
- `doubleclick`: `char con` `e = 2.718281828459045` (approx) `con` An approximation to the base of the natural logarithms.
- `encode`: `text→text` `dat` A not easily invertible function.
- `end`: `char con` The end-of-file character. It is greater than all letters, digits, punctuation marks, `space`, `tab`, and `newline`.
eval: text → all dat  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 dat  A function that says whether its argument is even or odd.

exec pro  A program with one text parameter. If the input represents a ProTem program, the execution is that of the represented program. It “unquotes” its argument. If applied to “x:= x+1”, the “x” refers to whatever x refers to at the location where exec “x:= x+1” occurs.

exp: com → com dat  An approximation to $e^x$.

false: bin con  A binary value.

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

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

floor: real → nat dat  floor r ≤ r < 1 + floor r

form: nat → nat → (nat+1) → real → text dat  Format a real number. form d e w r is a text representing real r with the final digit rounded. d is the number of digits after the decimal point; if d=0 the point is omitted. e is the number of digits in the exponent; if e>0 the decimal point will be placed after the first significant digit; if e=0 the “×10↑” is omitted and the decimal point will be placed as necessary. w is the total width; if w is greater than necessary, leading blanks are added; if w is less than sufficient, the text contains stars.

form 4 1 12 pi = “3.1416×10↑0”. form 2 0 6 (–pi) = “–3.14”. form 0 0 3 5 = “5”. form 0 0 0 3 (–5) = “–5”. form 0 0 2 123 = “***”.

formnum: text dat  A text format for numbers. It is useful for reading a number from a text channel. The number may be preceded by spaces.

g unit con  A unit representing mass in grams.
i = sqrt (–1) con  An imaginary number.
in: com → real dat  The imaginary part of a complex number.
index = text → * nat dat  A signal to the implementation that the string will be used only as an index to the indicated structure.

int dat  The integers.

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

lb: §(r: real → r>0) → real dat  An approximation to the binary logarithm (base 2).
ln: §(r: real → r>0) → real dat  An approximation to the natural logarithm (base e).
log: §(r: real → r>0) → real dat  An approximation to the common logarithm (base 10).

m unit con  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 cha  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 dat  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 con  The maximum representable integer (machine dependent).

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

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

mod: real → §(r: real → r>0) → real dat  mod a d is the remainder when a is divided by d.

$(0 \leq \text{mod } a \ d < d) \land (a = \text{div } a \ d \times d + \text{mod } a \ d)$
movie = *pic dat
nat dat: The natural numbers.
natchar: char nat -> char dat A one-to-one function with inverse charnat.
newline: char con The return or newline character.
nil con: The empty string.
null con: The empty bunch.
numtext: com -> text dat A text representation of a number. See also form.
odd: int -> bin dat A one-to-one function with inverse charnat.
ok pro: A program whose execution does nothing and takes no time.
ord = real, char, bin, [*ord] dat: The ordered type, for which < > ≤ ≥ are defined.
pi = 3.141592653589793 (approximately) con: An approximation to the ratio of a circle's circumference to its diameter.
pi = [x*[y*(0..z)]], dat: where x is the number of pixels in the horizontal dimension, y is the number in the vertical dimension, and z is the number of pixel values.
pre: char -> char con: The character predecessor function.
printer?! text cha: To the printer, it is an input channel; to all other programs, it is an output channel.
randomNat: nat -> nat -> nat dat: A reasonably uniform function, dependent on a hidden variable, over the interval from (including) the first argument to (excluding) the second argument.
randomNatInit pro: A program with one constant natural parameter. Its execution assigns a hidden variable to the natural value.
randomNatNext pro: A program. Its execution assigns a hidden variable to the next value in a random sequence.
randomReal: real -> real -> real dat: A reasonably uniform function, dependent on a hidden variable, over the interval between the arguments.
randomRealInit pro: A program with one constant real parameter. Its execution assigns a hidden variable to the real value.
randomRealNext pro: A program. Its execution assigns a hidden variable to the next value in a random sequence.
rat dat: The rational numbers.
re: com -> real dat: The real part of a complex number.
real dat: The real numbers.
round: real -> int dat: r-0.5 ≤ round r < r+0.5
s unit con: A unit representing time in seconds.
screen?! text cha: To the screen, it is an input channel; to all other programs, it is an output channel.
session: text dat: A text data name whose value is all keystrokes on channel keys since the start of a session.
sign: real -> (-1, 0, 1) dat
sin: real -> §(r: real -> -1 ≤ r ≤ +1) dat: An approximation to a trigonometric function.
sinh: com -> com dat: An approximation to a hyperbolic function.
stop pro: A program whose execution does nothing and takes forever so that no computation can follow.
subst: all -> all -> *all -> *all dat: 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 con: The character successor function.
tab: char con:
tan: (§(r: real -> -∃(i: int -> r = (2*i + 1)*pi))) -> real dat: An approximation to a trigonometric function.
tanh: \text{com} \rightarrow \text{com} \ \text{dat} \quad \text{An approximation to a hyperbolic function.}

text = *\text{char} \ \text{dat}

textnum: \text{text} \rightarrow \text{com} \ \text{dat} \quad \text{If the argument represents a number, possibly preceded by spaces, the result is the represented number.}

texttime: \text{text} \rightarrow (\text{int} \times \text{s}) \ \text{dat} \quad \text{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}

\text{texttime} \quad \text{“1947 September 16 at 19:24 UTC”} = -68675760\times s.

time?\!\text{real} \times \text{s} \ \text{cha} \quad \text{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: (\text{real} \times \text{s}) \rightarrow \text{text} \ \text{dat} \quad \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,}

\text{timetext} \quad (-68675760\times s) = \text{“1947 September 16 at 19:24 UTC”}

trim: \text{text} \rightarrow \text{text} \ \text{dat} \quad \text{A text formed from the argument by removing all leading and trailing space, \tab, and \arrayspace characters.}

true: \text{bin} \ \text{con} \quad \text{A binary value.}

wait \ \text{pro} \quad \text{A program with one constant parameter of type \text{real} \times \text{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 \text{await} and \text{time} and \text{s}.}
Example Program

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

  ` 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 definitions

new km : 1000×m. new h : 60×60×s. ` kilometer and hour
new maxaccel : 1.5×m/s/s. ` maximum deceleration = –maxaccel
new speedlimit : 60×km/h. ` speed limit is 60 km/h everywhere
new cushion : 1×s. ` reaction time for all porters
new impatience : 10/s. ` acceleration factor
new maxdistance : ceil (speedlimit↑ 2 / (2×maxaccel)). ` max search distance ahead
new numporters : 120.
new numboxes : 7480.
new visualdelaytime : 0.5×s. ` for human viewing

new porter. ` so porter can be indexed before it is defined
new box: [numboxes * ((“ahead left”, “ahead right”, “behind left”, “behind right”) → index “box”
  l“beside” → index “box”
  l“above” → index “porter”, numporters
  l(“x”, “y”) → nat )] ` box position on screen
  := [numboxes * ((“ahead left”, “ahead right”, “behind left”, “behind right”) → 0
  l“beside” → 0
  l“above” → numporters ` indicates no porter above
  l(“x”, “y”) → 0 ].

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

new draw do ⟨b: nat → {c: (“grey”, “blue”, “red”) → UNFINISHED}) od. ` end of draw
  ` draws a box at screen position (box b “x”) (box b “y”) of color c.
  ` “grey” means no porter present, “blue” means porter present, “red” means crash
  ` UNFINISHED because graphical output has not yet been designed

  ` end of definitions, start of initialization
```
ProTem

new x: 0; numboxes := 0. ` for input of box number
for b := 0; numboxes
    do screen! “What box is ahead-left of box”; numtext b; “?”.
        keys? !screen. x := textnum keys.
        box := (b; “ahead left”) → x | (x; “behind left”) → b | box.
    screen! “What box is ahead-right of box”; numtext b; “?”.
        keys? !screen. x := textnum keys.
        box := (b; “ahead right”) → x | (x; “behind right”) → b | box.
    screen! “What box is beside box”; numtext b; “?”.
        keys? !screen. box := (b; “beside”) → textnum keys | box.
    draw x “grey” od. ` default; may be changed below

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

old x.

randomNatInit 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: realtime := 10**38. ` 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 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] | fi | fi.
    boxesToDo := boxesToDo; [box b “ahead left”; 1|m].
    if box b “ahead left” + box b “ahead right”
    then boxesToDo := boxesToDo; [box b “ahead right”; 1|m] fi.
old b. old bb.

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

` using boxesToDo calculate accel for porter p

nextbox do new b:= (boxesToDo↓0) 0.`the box we are looking at
new d:= (boxesToDo↓0) 1.`its distance ahead of porter p
boxesToDo:= boxesToDo↓(1;..↔boxesToDo).
if d≤maxdistance
then new desiredspeed = `according to porter pa
    ⟨pa: (index “porter”, numporters) →
      if pa=numporters then speedlimit
      else (    sqrt (    porter pa “speed”↑2 + 2×maxaccel×d
                     + (maxaccel×cushion)↑2 )
                 − maxaccel×cushion ) ∧ speedlimit fi ).
    accel:= ( ( ( desiredspeed (box b “above”)
                     ∧ desiredspeed (porter (box b “beside”) “above”))
                − porter p “speed”)
            × impatience)
      ∨ −maxaccel ∧ maxaccel.
if box b “above” = numporters = porter (box b “beside”) “above”
then ` add boxes ahead to queue and continue
  boxesToDo:= boxesToDo; [box b “ahead left”; d+1×m].
  if box b “ahead left” ≠ box b “ahead right”
  then boxesToDo:= boxesToDo; [box b “ahead right”; d+1×m] fi.
nextbox
else if ↔boxesToDo > 0 then nextbox fi fi fi od.
old boxesToDo.

` using accel, move porter p ahead one box
new b: index “box”:= porter p “below”.
box:= (b; “porter”) → numporters | box. draw b “grey”.
randomNatNext.
b:= box b if randomNat 0 2 = 0 then “ahead left” else “ahead right” fi.
if box b “porter” < numporters then draw b “red”. stop fi. ` crash
porter:= (p; “below”) → b l porter. box:= (b; “above”) → p l 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)
l(p; “speed”) → speed
l porter.

await (iterationstarttime+visualdelaytime).
old speed. old accel. old p. old iterationstarttime.
infiniterloop od od `end of simport
Another Example Program

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

`output: screen
`use: even odd nat div bin false true numtext

new shl = 〈n: nat ⇝ (m: nat ⇝ ` shift n left m places; n×2^m
result r: nat:= n do for i:= 0..m do r:= r×2 od od〉.

new shr = 〈n: nat ⇝ (m: nat ⇝ ` shift n right m places; floor (n×2^-m) or div n (2^m)
result r: nat:= n do for i:= 0..m do r:= div r 2 od od〉.

new gcd = 〈a: (nat+1) ⇝ (b: (nat+1) ⇝ ` greatest common divisor of a and b
if a=b then a else if a<b then gcd a (b–a) else gcd (a–b) b fi fi〉.

new norm do (num::: (nat+1) ⇝ (denom:: (nat+1) ⇝ ` normalize num/denom
  new g:= gcd num denom.
  num:= num/g. denom:= denom/g〉 od.

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

for length:= 1..15
do for string:= 0..(shl 1 length) ` each string of that length
  do for quote:= 0..(shl length) ` each quote position (at least one bit to left of quote)
    do if even (shr string (length–1)) = even (shr string (quote–1)) ` roll-normalized
      then if ` repeat-normalized
        result repeatnorm: bin:= true
        do new len: nat = div (length–quote) 2. ` the length of the possibly repeating part
          trythislen do if len>0 ` 1 ≤ len ≤ (length–quote)/2
            then new extract = 〈i: nat ⇝ (l: nat ⇝ ` index i length l
              shr string i–shl (shr string (i+l)) l 〉
            new ex:= extract quote len.
            if ` the negative part is a repetition (twice or more) of ex
              result r: bin = true
              do new i: nat:= quote+len. ` i+len ≤ length
                iloop do new ey:= extract i len.
                if ex=ey then i:= i+len. ` i≤length
                  if i<length
                    then iloop
                  else r:= false fi fi
              else r:= false fi fi od od
            then repeatnorm:= false
            else len:= len–1. trythislen fi fi od od
          then for point:= 0..length+1 ` each point position (right end, interior, left end)
            do if ` the rightmost bit is 1 or it's to the left of quote or point
              odd string v quote=0 v point=0
then ` convert to numerator/denominator

new num: nat:= shl string (length–quote) – string
             – shl (shr string quote) length.

if num<0 then num:= –num fi.

new denom: nat:= shl (shl 1 (length–quote) – 1) point.

norm num denom.

` update statistics

count:= count+1. qlen:= qlen+length.
rlen:= rlen+1. ` for the sign

loop do num:= div num 2. rlen:= rlen+1.
    if num>0 then loop fi od.

loop do denom:= div denom 2. rlen:= rlen+1.
    if denom>0 then loop fi od fi od fi od od od.

screen! “In ”; numtext count; “ examples, quote average length = ”;

numtext (qlen/count); “, num/denom average length = ”; numtext (rlen/count)

old shl. old shr. old gcd. old norm. old count. old qlen. old rlen
**LL(1) Grammar**

In this grammar, for each nonterminal, every production except possibly the last begins with a different terminal. So director sets are not needed, and that's a special case of LL(1) that deserves its own name; I suggest LL(1/2). To parse a program, the parse stack begins with only the program nonterminal on it, and ends empty with no more input. However, ProTem functions as an operating system, parsing and executing each sequent in turn. So the parse stack begins with sequent on top, and \( \ll \) below it. When the stack is empty, the sequent is executed, the parse stack is reinitialized, and parsing resumes.

```
program  sequent aftersequent
sequent   phrase afterphrase
aftersequent   . program
      empty
phrase       new name afternewname
            old name
      do program od arguments
    if data then program elsepart fi arguments
      for name := data do program od
   ⟨ name parameterkind primary → program ⟩ arguments
      name aftername
elsepart      else program
      empty
parameterkind :
                          ::
                          !
                          ?
aftername    ::= data
                      ! data
                     ? echoordata
      do program od
arguments
echoordata   ! name
              data
afterphrase   \( \ll \) sequent
      empty
afternewname : data ::= data
                = data
                := data
                ? ! data
      do program od
```
unit
dictionary
name
empty
data
comparand aftercomparand
comparand
element afterelement
element
item afteritem
item
term afterterm
term
factor afterfactor
factor
# factor
– factor
~ factor
+ factor
? factor
☐ factor
\ factor
\ factor
* factor
& factor
$ factor
↔ factor
primary afterprimary
primary
number
text
if data then data else data fi arguments
result name : data := data do program od arguments
{ data }
[ data ] arguments
( data ) arguments
⟨ name : primary → data ⟩ arguments
name arguments
arguments
number arguments
text arguments
if data then data else data fi arguments
result name : data := data do program od arguments
{ data }
[ data ] arguments
( data ) arguments
⟨ name : primary → data ⟩ arguments
name arguments
empty
A name control procedure is responsible for classifying names. For efficiency, the productions (except possibly the last) for each nonterminal should be placed in order of frequency. The following nonterminals have only one production each, so they can be eliminated: program sequent data comparand element item term.
LR(0) Grammar

The following grammar has no reduce-reduce choices and no shift-reduce choices. It has shift-shift choices. Such a grammar is commonly called LR(0), but it shouldn't be, because a shift action pushes an input symbol onto the parse stack, and therefore a shift action depends on the input symbol. It is a special case of LR(1) that deserves its own name, but not LR(0); I suggest LR(1/2). To parse a program, the parse stack begins empty, and ends with only the program nonterminal on it and no more input. However, ProTem functions as an operating system, parsing and executing each sequent in turn. So the parse stack begins empty, and ends with . on top and sequent below it. The sequent is executed, the parse stack is reinitialized, and parsing resumes.

```
program
  sequent
  program . sequent

sequent
  phrase
  sequent || phrase

phrase
  new name : data := data
  new name := data
  new name = data
  new name do program od
  new name ? ! data
  new name unit
  new name dictionary
  new name name
  new name
  old name
  name := data
  name ! data
  name ? data
  name ? ! name
  name do program od
  if data then program fi
  if data then program else program fi
  for name := data do program od
  do program od
  procedure

procedure
  ⟨ name : primary → program ⟩
  ⟨ name :: primary → program ⟩
  ⟨ name ! primary → program ⟩
  ⟨ name ? primary → program ⟩
  procedure argument
  name

data
  data = comparand
  data ≠ comparand
  data < comparand
  data > comparand
  data ≤ comparand
```
\( \text{data} \geq \text{comparand} \)
\( \text{data} : \text{comparand} \)
\( \text{data} \in \text{comparand} \)
\( \text{data} \subseteq \text{comparand} \)
\( \text{comparand} \)

\( \text{comparand} , \text{element} \)
\( \text{comparand} , \ldots , \text{element} \)
\( \text{comparand} \mid \text{element} \)
\( \text{comparand} \triangleleft \text{data} \triangleright \text{element} \)
\( \text{element} \)

\( \text{element} ; \text{item} \)
\( \text{element} ; \ldots , \text{item} \)
\( \text{element} \cdot \text{item} \)
\( \text{item} \)

\( \text{item} + \text{term} \)
\( \text{item} - \text{term} \)
\( \text{item} \uparrow \text{term} \)
\( \text{item} \cup \text{term} \)
\( \text{term} \)

\( \text{term} \times \text{factor} \)
\( \text{term} / \text{factor} \)
\( \text{term} \wedge \text{factor} \)
\( \text{term} \vee \text{factor} \)
\( \text{term} \Delta \text{factor} \)
\( \text{term} \nabla \text{factor} \)
\( \text{term} \cap \text{factor} \)
\( \text{factor} \)

\( \text{factor} + \text{factor} \)
\( \text{factor} - \text{factor} \)
\( \text{factor} \cdot \text{factor} \)
\( \text{factor} \div \text{factor} \)
\( \text{factor} \wedge \text{factor} \)
\( \text{factor} \vee \text{factor} \)
\( \text{factor} \Delta \text{factor} \)
\( \text{factor} \nabla \text{factor} \)
\( \text{factor} \cap \text{factor} \)
\( \text{factor} \cup \text{factor} \)
\( \text{factor} \rightarrow \text{factor} \)
\( \text{factor} \uparrow \text{factor} \)
\( \text{factor} \downarrow \text{factor} \)
\( \text{factor} \)
\( \text{factor} \)
\( \text{factor} \)
A name control procedure is responsible for classifying names.