ProTem

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.

Symbols

ProTem has 12 keywords, plus 3 classes of symbols, plus 55 other symbols. Altogether they are:

```plaintext
if  then  else  fi  new  old  for  do  od  result  open  close
number  text  name
: :: := = == + < > ≤ ≥ √ ! ? , ; . || \ [ ] ( ) { } \[ ] \{ \} \[ ]\{ \}
```

A number is formed as one or more decimal digits, followed optionally 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. We shall see that numbers can be negated, added, subtracted, scaled, and so on.

A text begins with a left-double-quote, continues with any number of any characters (but a double-quote (left or right) must be repeated), and concludes with a right-double-quote. Here are four examples.

```
"abc" "don't" "Just say "no"." ""
```

A name is either simple or compound. A simple name begins with a letter and continues with any number of letters and digits, except that keywords cannot be names. A compound name is composed of one or more simple names joined with underscore characters. Here are some examples.

```
simple names: x A1 george refStack
compound name: Hehner_grammars_ProTem
```

A comment begins with a % that is not in a text, and ends at the end of a line.

The following standard keyboard symbols are unused in ProTem: `^` `&` `' `\`

Some of the ProTem symbols are not found on standard 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 size use \`
for \ use size
for element use \`
for \ use size
for element use \`
for " use "
for " use "
```

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

There are 24 ways of forming a program, and 53 ways of expressing data. (An LL\(^{1/2}\)) grammar and an LR\(^{1/2}\) grammar are at the end of this document.) Here are the ways of expressing data. To the right of each there are examples and explanations and pronunciations.

<table>
<thead>
<tr>
<th>expression</th>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>0 1.2</td>
<td></td>
</tr>
<tr>
<td>– data</td>
<td>minus, negation</td>
<td></td>
</tr>
<tr>
<td>data + data</td>
<td>plus, addition</td>
<td></td>
</tr>
<tr>
<td>data – data</td>
<td>minus, subtraction</td>
<td></td>
</tr>
<tr>
<td>data × data</td>
<td>times, multiplication</td>
<td></td>
</tr>
<tr>
<td>data / data</td>
<td>by, division</td>
<td></td>
</tr>
<tr>
<td>data ↑ data</td>
<td>to the power, exponentiation</td>
<td></td>
</tr>
<tr>
<td>data ∧ data</td>
<td>minimum, conjunction</td>
<td></td>
</tr>
<tr>
<td>data ∨ data</td>
<td>maximum, disjunction</td>
<td></td>
</tr>
<tr>
<td>data Δ data</td>
<td>negation of minimum, nand</td>
<td></td>
</tr>
<tr>
<td>data ∇ data</td>
<td>negation of maximum, nor</td>
<td></td>
</tr>
<tr>
<td>data = data</td>
<td>equals, equation</td>
<td></td>
</tr>
<tr>
<td>data ≠ data</td>
<td>differs from, discrepancy</td>
<td></td>
</tr>
<tr>
<td>data &lt; data</td>
<td>less than, strict implication</td>
<td></td>
</tr>
<tr>
<td>data &gt; data</td>
<td>greater than, strict reverse implication</td>
<td></td>
</tr>
<tr>
<td>data ≤ data</td>
<td>less than or equal to, implication</td>
<td></td>
</tr>
<tr>
<td>data ≥ data</td>
<td>greater than or equal to, reverse implication</td>
<td></td>
</tr>
<tr>
<td>data , data</td>
<td>bunch union</td>
<td></td>
</tr>
<tr>
<td>data , . . data</td>
<td>from(including)...to(excluding)</td>
<td></td>
</tr>
<tr>
<td>data * data</td>
<td>bunch intersection</td>
<td></td>
</tr>
<tr>
<td>data : data</td>
<td>bunch inclusion</td>
<td></td>
</tr>
<tr>
<td>∈ data</td>
<td>bunch size</td>
<td></td>
</tr>
<tr>
<td>; data</td>
<td>string catenation</td>
<td></td>
</tr>
<tr>
<td>; . . data</td>
<td>from(including)...to(excluding)</td>
<td></td>
</tr>
<tr>
<td>↓ data</td>
<td>string indexing</td>
<td></td>
</tr>
<tr>
<td>&lt; data &gt; data</td>
<td>string modification</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>“abc”</td>
<td></td>
</tr>
<tr>
<td>↔ data</td>
<td>string length</td>
<td></td>
</tr>
<tr>
<td>* data</td>
<td>definite repetition</td>
<td></td>
</tr>
<tr>
<td>* data</td>
<td>indefinite repetition</td>
<td></td>
</tr>
<tr>
<td>[ data ]</td>
<td>list</td>
<td></td>
</tr>
<tr>
<td># data</td>
<td>list length</td>
<td></td>
</tr>
<tr>
<td>+ data</td>
<td>list catenation</td>
<td></td>
</tr>
<tr>
<td>data @ data</td>
<td>list index, function application, composition</td>
<td></td>
</tr>
<tr>
<td>data @ data</td>
<td>pointer indexing</td>
<td></td>
</tr>
<tr>
<td>{ data }</td>
<td>set</td>
<td></td>
</tr>
<tr>
<td>~ data</td>
<td>set content</td>
<td></td>
</tr>
<tr>
<td>∈ data</td>
<td>element of a set</td>
<td></td>
</tr>
<tr>
<td>⊆ data</td>
<td>subset</td>
<td></td>
</tr>
<tr>
<td>∪ data</td>
<td>set union</td>
<td></td>
</tr>
<tr>
<td>∩ data</td>
<td>set intersection</td>
<td></td>
</tr>
<tr>
<td>↓ data</td>
<td>power</td>
<td></td>
</tr>
<tr>
<td>$ data</td>
<td>set size</td>
<td></td>
</tr>
</tbody>
</table>
~ data
\langle \text{name} : \text{data} \rightarrow \text{data} \rangle
\text{data} \rightarrow \text{data}
\square \text{data}
data \mid \text{data}
\text{name}
\check/ \text{name}
\text{if data then data else data fi}
do \text{program result data od}
( \text{data} )

Next we have the ways of forming a program.

\text{new name} : \text{data}
\text{new name} = \text{data}
\text{new name} :: \text{program}
\text{new name} !
\text{new name} == \text{name}
\text{new name}
\text{old name}
\text{open name}
\text{new open name}
\text{close name}
\text{name} := \text{data}
\text{name} ! \text{data}
\text{name} ? \text{data}
\text{name} !
\text{name} ?
\text{name} :: \text{program}
\langle \text{name} : \text{data} \rightarrow \text{program} \rangle
\text{program data}
\text{program} \cdot \text{program}
\text{program} \parallel \text{program}
\text{if data then program else program fi}
\text{for name} := \text{data} \text{do program od}
do \text{program od}

There is a precedence among the forms of program. It is
0. := ! ? \::\  \text{if fi do od}
1. \::\  \parallel

On level 1, the order is left-to-right. Program parentheses \text{do od} can always be used to group programs differently. The program
\text{new A}:: \text{B}. \text{C} \parallel \text{D}. \text{E}
when fully parenthesized, becomes
\text{do new A}:: \text{B od}. \text{do do C} \parallel \text{D od}. \text{E od}

A program form beginning with \text{new} , \text{old} , \text{open} , or \text{close} must be sequentially composed with a following program, or be followed by \text{result} .

Here is the order of evaluation of data operators.
0. number text name ( ) [ ] { } if fi do od
1. juxtaposition @ left-to-right
2. + – # ~ √ □ * → ↑ ↓ prefix + – # ~ √ □ * infix * → ↑ ↓ right-to-left
3. × / ∩ ∧ ∨ Δ ∨ infix / left-to-right
4. + – + ∪ infix – left-to-right
5. ; ;.. infix
6. , ,.. | ☐ ☐ infix < ☐ left-to-right
7. = + < > ≤ ≥ infix continuing

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

Whenever “data” appears in an alternative for “program”, the most general form of data is intended, with three exceptions. When a program is argumented, the argument must be on precedence level 0; therefore p a b means (p a) b. In a procedure and in a for-loop, the parameter/index type must be on precedence level 0.

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

Data

ProTem's basic data are numbers, characters, and binary values. ProTem's data structures are bunches, strings, lists, and functions. We refer to each of these (numbers, characters, binary values, bunches, strings, lists, and functions) as a type.

Numbers

In addition to the number symbols, there are predefined names of numbers such as pi (an approximation to the ratio of a circle’s circumference to its diameter), e (an approximation to the base of the natural logarithms), and i (the imaginary unit, or square root of -1). In addition to the 1-operand prefix operators + and – and the 2-operand infix operators + – × / ↑ there are predefined functions such as abs, exp, log, ln, sin, cos, tan, ceil, floor, round, re, im, sqrt, div, and mod (see Predefined Names). Division of integers, such as 1/2, may produce a noninteger. Exponentiation is 2-operand infix ↑; for example, 1.2×10↑3 (one point two times ten to the power three). The operators ∧ and ∨ are minimum (top is narrow, does not hold water) and maximum (top is wide, holds water) respectively.

In ProTem, subsets of numbers are not considered disjoint types. A natural number is an integer number; an integer number is a real number; a real number is a complex number.

Characters

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 four predefined character names: backspace, tab, return, and end (the end-of-file character). The operators suc and pre give the successor and predecessor respectively.
Binary Values

There are two predefined binary names in ProTem: true, and false. Negation is –, conjunction is \( \land \), disjunction is \( \lor \), nand is \( \Delta \), nor is \( \nabla \).

The infix 2-operand operators \( = \) and \( \neq \) apply to all data in ProTem with a binary result; the two operands may even be of different types. The order operators \( < \), \( > \), \( \leq \), \( \geq \) apply to real numbers, to characters, to binary values, to strings of ordered items, and to lists of ordered items, with a binary result; the two operands must be of the same type. In the binary order, false is below true, so \( \leq \) is implication. The 3-operand operator \( \text{if } x \text{ then } y \text{ else } z \text{ fi} \) has binary operand \( x \), but \( y \) and \( z \) are of arbitrary type.

Bunches

There are several predefined bunch names, namely:

- null - empty
- nat - all natural numbers: \( 0, 1, 2, ... \)
- int - all integer numbers: ...–2, –1, 0, 1, 2, ...
- real - all real numbers
- com - all complex numbers
- char - all characters
- bin - both binary values: true, false
- text - all texts (character strings)
- 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 element. For example, the number \( 2 \) is an elementary bunch, or synonymously, an 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” (but not “x through y”)

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

If \( A \) and \( B \) are bunches, then

\[ A : B \]  “A is included in B”

is binary.

Bunches are equal if and only if they consist of the same elements, without regard to order or multiplicity.

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

\[ –(3, 5) = –3, –5 \]
\[(2, 3)+(4, 5) = 6, 7, 8\]

This makes it easy to express the plural naturals \((nat+2)\), the even naturals \((nat\times2)\), the square naturals \((nat^2)\), the natural powers of two \((2^n)\), 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 example, \(\{0, 2, 5\}\), or \(\{0..100\}\), or \(\{null\}\), or \(\{nat\}\). The inverse of set formation is \(\sim\). For example, \(\sim\{0, 1\} = 0,1\). The size of a set is \(\#\). For example, \(#\{0, 1\} = 0,1\) and \(#\{null\} = 0\). The element, subset, union, and intersection operators \(\in \subseteq \cup \cap\) are as usual. The power operator \(\wr\) takes a bunch as operand and produces all sets that contain only elements of the bunch. For example, \(\wr\{0, 1\} = \{null\}, \{0\}, \{1\}, \{0, 1\}\).

**Strings**

Just as bunches are unpackaged sets, so strings are unpackaged lists. 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 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 example, \(0;..10\) is a string consisting of the first ten natural numbers, and \(5;..5\) is the empty string.

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 \(S\at{n}\rightarrow{i}\) is a string like \(S\) except that item \(n\) is \(i\). For example, \((3; 5; 9)\at{2}\rightarrow8 = 3; 5; 8\).

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

\[
\text{“abc”} = \text{“a”}; \text{“b”}; \text{“c”}
\]

\[
\text{“He said “Hi””} = \text{“H”}; \text{“e”}; \text{“i”}; \text{“s”}; \text{“a”}; \text{“d”}; \text{“i”}; \text{“e”}; \text{“r”}; \text{“n”}; \text{“H”}; \text{“i”}; \text{“e”}; \text{“r”}; \text{“n”}; \text{“.”}
\]

\[
\text{“abcdefghij”} \downarrow(3;..6) = \text{“def”}
\]

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, we have
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
\[ n \ast S \quad \text{“}n\text{ copies of } S\text{” or “}n S\text{'s”} \]
is a string, and
\[ \ast S \quad \text{“strings of } S\text{” or “any number of } S\text{'s”} \]
is a bunch of strings. For examples,
\[ 3 \ast 5 = 5;5;5 \]
\[ 5 = \text{nil}, 5;5, 5;5;5, 5;5;5;5, \ldots \]
The $\ast$ operator distributes over bunch union, but in its left operand only.
\[ \text{null} \ast 5 = \text{null} \]
\[ (2,3) \ast 5 = (2\ast5),(3\ast5) = 5;5, 5;5;5 \]
Using this semi-distributivity, we have
\[ \ast a = \text{nat}\ast a \]

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, and let $n$ be a natural number.
\[ #L \quad \text{“}L\text{’s length”} \]
\[ \sim L \quad \text{“}L\text{’s content”} \]
\[ Ln \quad \text{“}L \text{ at } n\text{”, “}L\text{ at index } n\text{”} \]
\[ L@p \quad \text{“}L \text{ at pointer } p\text{”} \]
\[ L+M \quad \text{“}L \text{ catenate } M\text{”, “}L \text{ join } M\text{”} \]
\[ LM \quad \text{“}L \text{ composed with } M\text{”} \]
\[ L|M \quad \text{“}L \text{ otherwise } M\text{”, “the selective union of } L \text{ and } M\text{”} \]
plus the comparisons $L=M$, $L+M$, $L<M$, $L>M$, $L\leq M$, $L\geq M$.
Here are some examples.
\[ #[0; 1; 2] = 3 \quad \text{(the number of items in a list)} \]
\[ \sim[0; 1; 2] = 0;1;2 \]
\[ [0;..10] 5 = 5 \quad \text{(indexing starts at zero)} \]
\[ [2; 3]; 4; [5; [6; 7]] ] @ (2; 1; 0) = 6 \]
\[ [0;..10] + [10;..20] = [0;..20] \]
\[ [10;..20] [3; 6; 5] = [13; 16; 15] \quad \text{(in general, } (L M)n = L(M n) .) \]
If a list is indexed with a structure, the result has the same structure. For example
\[ [10; 20] [2; (3, 4); [5; [6; 7]]] = [12; (13, 14); [15; [16; 17]]] \]
By using the $@$ operator, a string acts as a pointer to select an item from within an irregular structure. If the list $L|M$ is indexed with $n$, the result is either $Ln$ or $Mn$ depending on whether $n$ is in the domain $(0,..,#L)$ of $L$. If it is, the result is $Ln$, otherwise the result is $Mn$.
\[ [10; 11] | [0;..10] = [10; 11; 2;..10] \]
Lists are equal if and only if they are the same length and corresponding items are equal. They are ordered lexicographically.
\[ [3; 5; 2] < [3; 6] \]
The list brackets $[ ]$ distribute over bunch union. For example,
\[ [0, 1] = [0], [1] \]

Thus \([10^{*}\text{nat}]\) is all lists of length 10 whose items are natural, and \([4^{*}[6^{*}\text{real}]]\) is all 4 by 6 arrays of reals.

**Functions**

Let \(p\) be a fresh name, let \(D\) be a bunch of items, and let \(B\) be an element (possibly using \(p\) as an element of \(D\)). Then

\[ \langle p: D \rightarrow B \rangle \]

is a function in parameter \(p\) with domain \(D\) and body \(B\). For example,

\[ \langle n: \text{nat} \rightarrow n+1 \rangle \]

“map \(n\) in \(\text{nat}\) to \(n+1\)”

is the successor function on the natural numbers.

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

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 \(\langle n: 2 \rightarrow 3 \rangle\). An example of its use is

\[ 1 \rightarrow 21 | [10; 11; 12] = [10; 21; 12] \]

We allow domains to be strings in the following circumstances.

\[ \text{nil} \rightarrow x \mid f = x \]

\[ (x,y) \rightarrow z \mid f = x \rightarrow (y \rightarrow z) \mid f \]

Thus, for example,

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

we have

\[ [3; 6; 2]; [3; 4; 5]] = [3; 4; 5] \]

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

\[ (f, g) (x, y) = f x, f y, g x, g y \]

Allowing the body of a function to be a bunch generalizes it to a relation. For example, \(\text{nat} \rightarrow \text{bin}\) can be viewed in any of the following three equivalent ways: it is a function (with unused and therefore omitted parameter) that maps each natural to \(\text{bin}\); it is a relation relating each natural to each binary value; it is all functions with domain \(\text{nat}\) and range \(\text{bin}\). As an example of the last view, we have

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

**Input Test**

If \(c\) is the name of an input channel, then

\[ \sqrt{c} \]

is a binary expression saying whether there is currently any unread input on channel \(c\).
Programmed Data

The value of

\[ \text{do program result data od} \]

is obtained by executing the program and then evaluating the data. We have not yet presented
programs, but the following example, which approximates \( e \), should give the idea.

\[ \text{do new sum: real. sum:= 1.} \]
\[ \quad \text{new term: real. term:= 1.} \]
\[ \quad \text{for } i := (1;..15) \text{ do term:= term/i. sum:= sum+term od} \]
\[ \text{result sum od} \]

There are no side effects. Suppose \( x \) is a natural variable with value 5. Then evaluation of

\[ \text{do } x := x+1 \text{ result } x \text{ od} \]
yields 6, but variable \( x \) remains unchanged.

Names and Dictionaries

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

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

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

Whenever a compound name is used, it is looked up as follows. The first simple name in the
compound name is looked up in the usual way (starting with the open dictionary that was opened
last). Its definition must be as a dictionary. The second simple name in the compound name is
looked up in this one dictionary (whether open or closed). If there is a third name in the compound
name, then the second name must be defined as a dictionary, and the third name is looked up in that
dictionary. And so on for subsequent names in a compound name.

Names are defined by language constructs beginning with \( \text{new} \). Whenever a simple name is
defined, its definition is written in the open dictionary that was opened last (it must not already be
there). When a compound name is defined, the last simple name in the compound name is placed in
the dictionary referred to by the compound name minus its last simple name.
Names can be removed from a dictionary with the keyword \texttt{old} (it must already be there). Further details and examples will be presented later.

\textbf{Programs}

About half of the program constructs are concerned with dictionaries: adding names \texttt{(new)}, deleting names \texttt{(old)}, opening a dictionary \texttt{(open)}, and closing a dictionary \texttt{(close)}. The other half are variable assignment, input, output, and a variety of ways of combining programs to form larger programs.

\textbf{Variable Definition}

Here is an example variable definition (declaration).

\begin{verbatim}
\texttt{new x: 0..10}
\end{verbatim}

This defines $x$ to be a variable assignable to any element in $0..10$, and initially assigned to an arbitrary element in that bunch. In other words, $x$ is defined so that $x: 0..10$ is always true, even initially. But the subsequent program cannot depend on any particular initial value. There is no such thing as “the undefined value” in ProTem. In a variable definition, the data after the colon is called the “type” of the variable. The type can be anything except the empty bunch. The type can depend on previously defined names, including variables.

If you want a variable to be defined with a specific initial value, just follow the definition with an assignment. Here are three examples.

\begin{verbatim}
\texttt{new s: [10*int]. s:= [10*0]}
\texttt{new t: text. t:= ""}
\texttt{new u: (0..20)*char. u:= "abc"}
\end{verbatim}

$s$ 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 second example, \texttt{text} is a predefined bunch equal to \texttt{*char}, so $t$ can be assigned to any text, and is initially assigned to the empty text. In the third 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 shown.

\textbf{Assignment}

Assignment is as usual; the data on the right must be an element in the type of the variable on the left. Here are two examples using the definitions of the previous section.

\begin{verbatim}
x:= 5
s:= 3 → 5 | s
\end{verbatim}

\textbf{Data Definition}

Data definition gives some data a name. Here are four examples.

\begin{verbatim}
\texttt{new size = 10}
\texttt{new range = 0..size}
\texttt{new piBy2 = pi / 2}
\texttt{new y = s x}
\end{verbatim}

The first three names are constants because their definitions use only constants (\texttt{pi} is a predefined constant). The fourth definition makes $y$ depend on variables $s$ and $x$ defined previously; the value of $y$ changes according to the values of $s$ and $x$ so that $y = s x$ is always true. We may call
s and x “independent variables”, and y a “dependent variable”. Notice the difference between this and

```
new z: s x
```

Here, z is defined as an independent variable whose type is a single value, namely, the value of s x when this definition is executed. It therefore has that value. Its value does not change when s and x change.

The next two examples define fact and div to be the factorial function and integer divisor function for natural numbers. They are both constants. Note the use of recursion.

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

new div = \(\langle a: nat → \langle d: (nat+1) →
  \begin{cases}
    0 & \text{if } a < d \\
    2 \times \text{div} (a/2) d & \text{if even } a \\
    1 + \text{div} (a–d) d fi 
  \end{cases}\)fi
```

A final example defines all binary trees with integer nodes.

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

Program Definition

Program definition gives a program a name. For example,

```
new switchends:: s:= 0 → (s 9) | 9 → (s 0) \| s
```

The double colon is pronounced “is”. 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. Definitions can be recursive.

One of the predefined program names is ok, whose execution does nothing. It is used, for example, with if. (See Conditional Program.)

Forward Definition

The “forward definition” is a notice that a definition will follow later. It is used, for example, when definitions are mutually recursive. (See Name Duration.)

Name Removal

Names are added to a dictionary with the keyword new, and they are removed from a dictionary with the keyword old. Even though a name may be removed from a dictionary, its definition will remain as long as there is an indirect way to refer to it. For example,

```
new s: [*all]. s:= [nil].
new push:: \langle x: all → s:= s \oplus [x]\).
new pop:: s:= s [0;..#s–1].
new top = s (#s–1).
new empty = s=[nil].
old s.
```

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

The syntax

```
new open d
```

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

```
open d
```

is used to open an existing but closed dictionary \( d \). The syntax

```
close d
```

is used to close an open dictionary.

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

```
new open random.
new big = 2↑31.
new rv: 0,..big.
new init: (seed: (0,..big) → rv:= seed).
new next: rv:= mod (rv × 5↑13) big.
new value = (from: real → (to: real → from + (to–from)×rv/big)).
old big. old rv.
close random.
```

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

```
random_init 123456789.
random_next.
screen! numtext (random_value 0 1).
```

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

```
open random.
init 123456789.
next.
screen! numtext (value 0 1).
```

We can get rid of a dictionary name \( d \) by saying

```
old d
```

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

Synonym Definition

The syntax

```
new a == b
```

makes name `a` a synonym of name `b`. The two names are now equivalent in every way. Synonym definition can be used to rename a variable or channel or dictionary as follows.

```
new newname == oldname. old oldname
```

It can be used to move a name from one dictionary to another as follows:

```
new toDict_name == name. old name
```

or as follows:

```
new name == fromDict_name. old fromDict_name
```
depending on which dictionary was opened last.

Output and Input

The channels screen, printer, keys, cursor, and time are predefined.

The program

screen! "Hi there"

sends the text “Hi there” to the screen. A string of outputs can be sent together

screen! “Answer = ”; numtext x; “ meters”

This is equivalent to

screen! “Answer = ”. screen! numtext x. screen! “ meters”

One integer of input is requested on channel keys by the program

every? int

If input is not yet available, it is awaited. When the input is received, it is referred to simply as keys.

Sequential Composition

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

Parallel Composition

For programs P and Q that assign to different variables, or different parts of a structured variable, the parallel composition of P and Q is denoted P||Q. Each program can use the variables assigned by the other, but all occurrences of variables assigned by the other program refer to their initial value. Parallel programs cannot affect each other through assignments to variables. For cooperation, programs can communicate with each other on channels defined for the purpose.

Channel Definition

The definitions

new c!
new d!

define c and d to be channels. They can be used, even without parallelism, just as buffers. For example,

c! 5. c! 7. c? int x:= c. c? int y:= c
assigns 5 to x and 7 to y. More often, channels are defined for communication between parallel programs.

\[
\text{do } c! 7. \text{ d? int } x:= d \text{ od }
\]

\[||\text{ do } d! 5. \text{ c? int } y:= c \text{ od}\]

The output in one program can be input in all parallel programs independently. However, two parallel programs cannot use the same channel for output.

Conditional Program

The if is as usual. There is no one-tailed if in ProTem; the ProTem equivalent is shown in the following example.

if x > y then x:= y else ok fi
An “assert” program is obtained according to the following example.

\[
\text{if } x > y \text{ then ok else screen! “appropriate error message”}. \text{ stop fi}
\]

**Named Programs**

A named program has the syntax

\[
\text{name :: program}
\]

The name is attached to the program (like a program definition), and the program is executed (unlike a program definition). The program name is known only within the program to which it is attached. One purpose of this naming is to make loops. Here is a two-dimensional search for a 5 in an \(n\times m\) array \(A\) of integers (that is, \(A: [n^* [m^* \text{int}]] \)).

```plaintext
new i: nat. i := 0.
tryThisI:: if i = n then screen! “5 does not occur in A.”
else new j: nat := 0.
tryThisJ:: if j = m then i := i + 1. tryThisI
else if A[i][j] = 5 then screen! “5 occurs at”; i; j
else j := j + 1. tryThisJ fi fi fi
```

The second example is a fast remainder program, assigning to natural variable \(r\) the remainder when natural \(a\) is divided by natural \(d\), using only addition and subtraction.

\[
r := a.
outerloop:: if r < d then ok
else new dd: nat := d.
innerloop:: do r := r – dd. dd := dd + dd.
if r < dd then outerloop else innerloop fi od fi
\]

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 \(f_n\) is the \(n\)th Fibonacci number, and it does so in \(\log n\) time.

\[
F:: \text{if } n = 0 \text{ then } x := 0. \ y := 1
\text{else if odd } n \text{ then } n := (n-1)/2. \ F. \ n := x. \ x := x \uparrow 2 + y \uparrow 2. \ y := 2 \times n \times y + y \uparrow 2
\text{else } n := n/2 – 1. \ F. \ n := x. \ x := 2 \times x \times y + y \uparrow 2. \ y := n \uparrow 2 + y \uparrow 2 + x \text{ fi fi}
\]

The following two lines are equivalent.

\[
P:: Q
\text{ do new } P:: Q. \ P \text{ od}
\]

**Controlled Program**

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

\[
\text{for } j := (0..n)
\text{ do for } i := (0..n)
\text{ do for } k := (0..n)
\quad \text{do } A := (i; k) \rightarrow (A \ i \ k \lor (A \ i \ j \land A \ j \ k)) \quad | \ A \ \text{od od od}
\]

The assignment can be restated as

\[
\text{if } A \ i \ j \land A \ j \ k \text{ then } A := (i; k) \rightarrow \text{true} \quad | \ A \ \text{else ok fi}
\]

if you prefer. The name being introduced by the \texttt{for} loop is known only within the loop body, and it is known there as a data name. It is not a variable, and so it is not assignable. We call it the \texttt{for} parameter.
For a second example, here is the sieve of Eratosthenes.

```plaintext
new n = 1000.
new prime: [n*bin]. prime:= [2*false; (n–2)*true].
for i:= (2; ceil (sqrt n))
do if prime i then
   for j:= (i; ceil (n/i)) do
      prime:= (i*j) → false \ prime od
else ok fi od
```

The `for` parameter is “by initial value”, so

```plaintext
for i:= (x; x) do x:= i+1 od
```

increases `x` by 1, not 2.

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

```plaintext
for i:= (0,..n) do A:= i → 0 \ A od
```

makes the first `n` items of `A` be 0, in parallel.

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

**Procedures**

A program can be given parameters, as in this example.

```plaintext
new transformX:: (magnification: num → (translation: num →
   x:= magnification×x + translation))
```

A program with one or more parameters is called a “procedure”. A procedure can be argumented in the same way that lists are indexed and functions are argumented. Here is a procedure with one parameter

```plaintext
new translateX:: transformX 1
```

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.

```plaintext
new magnifyX:: (magnification: num → transformX magnification 0)
```

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

```plaintext
magnifyX 3
transformX 3 0
```

A procedure's parameter is known only within the procedure body, and it is known there as a data name. It is not a variable, and so it is not assignable. It is “by initial value”, so

```plaintext
⟨i: int → x:= i, y:= i⟩ (x+1)
```

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

**Scope**

In general, scopes are limited by `do od` and ⟨⟩ brackets.
A name introduced by the keyword **new** must be fresh, i.e. not defined since the previous **do** or **〈**. Its scope extends from its definition to the next **od** or **〉**. But it may be covered by a definition in a more local scope. For example, letting $A, B, C, \ldots$ stand for arbitrary program forms, in

$$
A. \textbf{new } x: \textit{int}. \ B. \textbf{do } C. \textbf{new } x: \textit{bin}. \ D \textbf{od}. \ E
$$

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

In a variable definition, the name being introduced cannot be used in the type and initialization. Its scope begins after that.

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

$$
\textbf{new } f = 3. \ \textbf{do } \textbf{new } f. \ \textbf{new } g = \cdots f \cdots g \cdots. \ \textbf{new } f = \cdots f \cdots g \cdots. \ B \textbf{ od}
$$

$f$ and $g$ are each defined in terms of both of them. Without the forward definition of $f$, $g$ would be defined in terms of the more permanent $f=3$.

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

$$
\textbf{new } x = 0. \ A. \ \textbf{old } x. \ B
$$

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 **do** or **〈**. After **old**, the name $x$ is again fresh and available for definition. However,

$$
\textbf{new } x = 0. \ \textbf{do old } x. \ A \textbf{ od}
$$

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

A name can be introduced without the keyword **new**, as a parameter of a procedure or function or **for** loop, or as the name of a named program. Any such name must be fresh within the most local scope, i.e. since the previous **do** or **〈**, just like a name introduced with the keyword **new**. Its scope extends only through the program or data to which it is attached, not beyond. After that, it is again fresh and available for definition.

The data form

$$
\textbf{do program result data od}
$$

constitutes a scope. The scope extends through the data as though the keyword **result** were a sequential composition. In the following example, the program part of a **result** expression is used purely to make a definition.

$$
\textbf{new } acc = (\textit{func} (\textit{com} \rightarrow \textit{com} \rightarrow \textit{com})) \rightarrow (\textit{list} : [*\textit{com}] \rightarrow

\begin{align*}
\text{if } & \#\textit{list} < 2 \\
\text{then } & \textit{list} \\
\text{else } & \textbf{new } c = acc \textit{func} (\textit{list} [0;..\#\textit{list}–1])
\end{align*}

\textbf{result } c^+ [\textit{func} (c (\#c–1)) (\textit{list} (\#\textit{list}–1))] \textbf{ od fi})
$$

When function $acc$ (accumulation) is applied to the addition function and a list, the result is the list of cumulative sums. For example,

$$
acc \langle x : \textit{com} \rightarrow \langle y : \textit{com} \rightarrow x + y \rangle \rangle [3; 5; 4; 4] = [3; 8; 12; 16]
$$

The syntax **do new ... result ... od** has exactly the effect of “let ... in ...” in some languages. Its purpose here is simply to introduce an abbreviation for a long expression that is used three times.
If a definition occurs outside all `do od` and `〈 〉` pairs, its scope ends only with an `old` . Its scope does not end with the end of a computing session, not even by switching off the power. Variables declared outside all parentheses serve as “files”. A predefined name cannot have its scope ended by `old` , but it can be obscured by a programmer's definition of the same name.

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

```
A. do B od. C
```

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

```
open d. do close d. A od
```

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

**Miscellaneous**

The ProTem equivalent of enumerated type is shown here.

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

The ProTem equivalent of the record type is as follows.

```
new person =   “name” → text
              | “age” → nat.
```

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

```
```

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

```
new file: *person. file:= nil.
```

and concatenate new records onto its end.

```
file:= file; p.
```

The efficiency of pointers is obtained through the use of three predefined names. The first is:

```
new index = text→nat
```

When applied to a text argument, it ignores its argument and yields the result `nat` . The use of `index` is a signal to the implementation that the natural number will be used only as an index into the list whose name is given by the text argument (and the implementation will check that this is so). For example,

```
new G: [* (“name” → text | “next” → (index “G”))] := [“name” → “zzzzz” | “next” → 0].
new first: index “G” := 0.
```

We can still assign `first` to a natural number, for example

```
first:= first+1
```

and similarly for the “next” field of each record of `G` . But we can use them only as indexes into `G` , for example

```
first:= G first “next”
G:= first → (“name” → “Aaron” | “next” → first) | G
G:= first; “name” → “George” | G
```
With this limited use, the implementation of these indexes can be a memory address. This way we obtain all the performance benefits of pointers without destroying the logic of our language.

The other two predefined names that give pointer efficiency are

\[
\begin{align*}
\text{new } \text{path} &= \text{text} \rightarrow \ast \text{nat} \\
\text{new } \text{backup} &= \langle \text{list} : \ast \text{nat} \rightarrow \sim \text{list} [0 ; \ldots ; \# \text{list} - 1] \rangle
\end{align*}
\]

Like \textit{index}, \textit{path} ignores its argument. The use of \textit{path} is a signal to the implementation that the string of natural numbers will be used only as a string of indexes into the structure whose name is given by the text argument (and the implementation will check that this is so). For example,

\[
\begin{align*}
\text{new } \text{tree} &= [\text{nil}, [\text{tree}; \text{all}; \text{tree}]]. \\
\text{new } t: \text{tree}. \ t &= [\text{nil}]. \\
\text{new } p: \text{path} \text{ “} t \text{”}. \\
\end{align*}
\]

To move \( p \) down to the left in the tree we reassign it this way:

\[
p := p ; 0
\]

and similarly to move it down to the right. To move it up, we just remove its final item

\[
p := \text{backup} [p]
\]

Indexing \( t \) with \( p \) yields a subtree of \( t \)

\[
t @ p
\]

and we can replace this subtree with tree \( s \) using the assignment

\[
t := p \rightarrow s | t
\]

We can express the information at the node indicated by \( p \) as

\[
t p 1 \quad \text{or} \quad t @ (p; 1)
\]

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

\[
t := p; 1 \rightarrow 6 | t
\]

We obtain the performance benefit of having \( p \) implemented as a string of addresses rather than as a string of natural numbers, without complicating the language.

Exception handling is provided by bunch union or by the \( | \) operator. For example,

\[
\text{new } \text{divide} = \langle \text{dividend}: \text{com} \rightarrow \langle \text{divisor}: \text{com} \rightarrow
\]

\[
\text{if } \text{divisor} = 0 \text{ then “} \text{zero divide} \text{” else dividend / divisor } \rangle
\]

We can state the type of this function as

\[
\text{com, “} \text{zero divide} \text{”}
\]

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

\[
\text{new } \text{weekday} = \langle d: (0; \ldots ; 7) \rightarrow 1 \leq d \leq 5 \rangle
\]

Then in the expression

\[
(\text{weekday} | \text{all} \rightarrow \text{“domain error”}) \ i
\]

if \( i \) fails to be an integer in the range \( 0; \ldots ; 7 \), 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.

\[
\text{inputchoice:} \quad \text{if } \sqrt{c} \text{ then } c?\text{int. } P
\]

\[
\text{else if } \sqrt{d} \text{ then } d?\text{int. } Q
\]

\[
\text{else inputchoice } \text{fi } \text{fi}
\]

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

% input: on these channels.
% output: on these channels.
% need: the values of these variables.
% assign: these variables.
% use: these data names
% call: these program names.
% refer: to these dictionaries.

We make them transitive through “use” and “call”, even without requiring the implementation to do
a transitive closure (it just checks the comments at the head of the used data names and called
program names).

A loophole that allows us to drop down into assembly language is provided by the predefined
procedure `asm`, which 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
`Pascal`, with one text parameter, whose execution is that of the Pascal fragment represented by the
argument.

**Other Issues**

**Object Orientation**

ProTem is not an object-oriented language. ProTem considers object orientation to be a
programming style, rather than a programming-language style, or collection of language features.
Object-oriented programming (as a style of programming) can be done in ProTem, and should be
done whenever it is helpful. Data structures, and the functions and procedures that access and
update them, can be defined together and given a consistent naming convention. If many objects of
the same type are wanted, they can be defined as items in a list. Or, if you prefer, they can be
instantiated by re-invoking the program that defines one of them.

**Documents**

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

```
new p: pic.
```

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

Another predefined name is `movie`, defined as \([*pic]\). The operations on movies are just those of
lists, such as catenation. To help in the creation of movies, one of the pixel values should be
“transparent”, and one of the operations on pictures should be overlaying one picture on another.
WE NEED a sound (musical) data type. WE ALSO NEED a way to combine all of these types in one document. WE ALSO NEED to be able to define regions of documents to be links, so that when clicked some action occurs.

Editing

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

Session

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

At the start of a session, the screen is clear and is divided into two regions; one region reflects what is input on channel key, and the other reflects output from the computer on channel screen. Input and output on separate channels are not mixed. Also at the start of a session, only the root dictionary is open and all passwords are required. A password will not be requested twice within the same session for the same dictionary.

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

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

Security

Any dictionary may contain a variable definition such as

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

If a dictionary contains a password variable, the text whose encoding is its value will be requested when an attempt is made to open the dictionary or to refer to its contents. Passwords belong to dictionaries, not to people. For example

```
new open readBarrier.
new password: nat. password:= encode “read code”.
new open writeBarrier.
new password: nat. password:= encode “write code”.
new it: real. it:= 17.2.
close writeBarrier.
new readonlyit = writeBarrier_it.
```
**close readBarrier.**

To use data name `readonlyit`, either by opening dictionary `readBarrier` or as `readBarrier_readonlyit`, you must know the password “read code”. This enables you to know the value of variable `it`, but not to change it. To change it, you must know a second password, “write code”.

**Error Recovery**

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

Most of the time undo can be implemented just by doing nothing; the nonvolatile memory (disk) contains the state as it was before the start of the previous program, and the volatile memory contains the current state, which is stored in the 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 the 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. copy:= session.
```

One then edits the keystroke file, 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 are:

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

**Predefined Names**

`abs: com→real`. Absolute value. For complex `x`,

```
abs x = sqrt (re x↑2 + im x↑2).
```

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

await. A program with one parameter of type \(5\times \text{nat}; \text{real}\). Its execution does nothing but takes time until the instant given by the argument. See time and wait.

backup: \([\ast \text{nat}] \rightarrow \ast \text{nat}\). backup \([s; i] = s\). For use with path.

bin = true, false.

calculus. A dictionary containing the following names.

\[ e = 2.718281828459045 \text{ (approx).} \]  An approximation to the base of the natural logarithms.
\[ \text{exp: real} \rightarrow \text{real}. \]  An approximation to \(e^x\). (The domain and range should probably include complex numbers, but I have forgotten my complex analysis.)
\[ \text{lb: } \$(r; \text{real} \rightarrow r>0) \rightarrow \text{real}. \]  An approximation to the binary logarithm (base 2).
\[ \text{ln: } \$(r; \text{real} \rightarrow r>0) \rightarrow \text{real}. \]  An approximation to the natural logarithm (base \(e\)).
\[ \text{log: } \$(r; \text{real} \rightarrow r>0) \rightarrow \text{real}. \]  An approximation to the decimal (common) logarithm (base 10).

ceil: \text{real} \rightarrow \text{int}. \ r \leq \text{ceil} r < r+1

char. The characters.

charint: \text{char} \rightarrow \text{int}. A one-to-one function with inverse intchar.

com. The complex numbers.

complex. A dictionary containing the following names.

\[ \text{arc: com} \rightarrow \$(r; \text{real} \rightarrow 0 \leq r < 2\pi). \]  An approximation to the angle or arc of a complex number.
\[ i = \text{sqrt}(-1). \]  The imaginary unit.
\[ \text{im: com} \rightarrow \text{real}. \]  The imaginary part of a complex number.
\[ \text{re: com} \rightarrow \text{real}. \]  The real part of a complex number.

comtext: \text{com} \rightarrow \text{text}  A textual representation of a complex number.

cursor! [\text{nat}; \text{nat}]. An input channel.

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

\[ \text{div: real} \rightarrow \$(r; \text{real} \rightarrow r>0) \rightarrow \text{int}. \]  \(\text{div a d}\) is the integer quotient when \(a\) is divided by \(d\).
\[ (0 \leq \text{mod a d} < d) \land (a = \text{div a d} \times d + \text{mod a d}). \]

code: text \rightarrow \text{nat}. A not easily invertible function. SHOULD IT BE text \rightarrow \text{text}.

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

eval: text \rightarrow \ast \text{all}. If the argument represents a ProTem data expression, the evaluation is that of the represented data. It “unquotes” its argument.

even: \text{int} \rightarrow \text{bin}.

dec. A program with one text parameter. If the argument represents a ProTem program, the execution is that of the represented program. It “unquotes” its argument.

false: bin. screen! false prints false

find: \ast \text{all} \rightarrow \text{nat}. If \(i\) is an item in \(L\), then find \(iL\) is the index of its first occurrence; if not, then find \(iL = \#L\).

fit: text \rightarrow \text{text}. If \(i\geq0\) then fit \(ti\) 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 \(i\leq0\) then fit \(ti\) 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: \text{real} \rightarrow \text{int}. floor \(r \leq r < 1 + \text{floor} r\)

form: \text{real} \rightarrow \text{nat} \rightarrow (\text{nat}+1) \rightarrow \text{text}. Format a real number. form \(r d e w\) is a text representing real \(r\) with the final digit rounded. \(d\) is the number of digits after the decimal point; if \(d=0\) the
point is omitted. \( e \) is the number of digits in the exponent; if \( e > 0 \) the decimal point will be placed after the first significant digit; if \( e = 0 \) the \( \times 10^0 \) 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.

\[
\begin{align*}
\text{form } & \pi 4 1 2 = "3.1416\times10^1". \\
\text{form } & (–\pi) 2 0 6 = "–3.14". \\
\text{form } & 5 0 0 3 = "5". \\
\text{form } & 1 2 3 0 0 2 = "*".
\end{align*}
\]

**hyperbolic.** A dictionary containing the following names. (The domains and ranges should probably include complex numbers, but I have forgotten my complex analysis.)

\[
\begin{align*}
\text{cosh: real} & \rightarrow \text{real}. \text{An approximation to a hyperbolic function.} \\
\text{sinh: real} & \rightarrow \text{real}. \text{An approximation to a hyperbolic function.} \\
\text{tanh: real} & \rightarrow \text{real}. \text{An approximation to a hyperbolic function.}
\end{align*}
\]

\[
\text{index = text} \rightarrow \text{nat}. \text{A signal to the implementation that the natural number will be used only as an index to the indicated list.}
\]

\textit{int.} The integers.

\textit{intchar: charint char} \rightarrow \text{char}. \text{A one-to-one function with inverse \textit{charint}.}

\textit{key!}. An input channel.

\textit{line! text.} An input channel giving the same input as \textit{key} except that the characters are gathered into texts, each text extending from one \textit{return} character to the next, not including the \textit{return} characters, and corrected according to the use of \textit{backspace} characters.

\textit{mail.} A CHANNEL, PERHAPS?

\textit{match: *all} \rightarrow \text{*all} \rightarrow \text{nat}. \text{If pattern occurs within subject, then match pattern subject is the index of its first occurrence. If not, then match pattern subject = ↔subject.}

\textit{mod: real} \rightarrow \text{§(r: real} \rightarrow \text{r>0)} \rightarrow \text{real. mod a d is the remainder when a is divided by d.}

\[
(0 \leq \text{mod a d < d}) \wedge (a = \text{div a d} \times d + \text{mod a d})
\]

\textit{movie = [*pic].}

\textit{nat.} The natural numbers.

\textit{nil.} The empty string.

\textit{null.} The empty bunch.

\textit{numtext: com} \rightarrow \text{text}. \text{A text representing the argument.}

\textit{odd: int} \rightarrow \text{bin.}

\textit{ok.} A program whose execution does nothing.

\textit{openlist: text.} The names of the open dictionaries in the order they were opened.

\textit{path = text} \rightarrow \text{*nat.} A signal to the implementation that the string will be used only as an index to the indicated list.

\textit{pi} = 3.141592653589793 (approx). An approximation to the ratio of a circle's circumference to its diameter.

\textit{pic} = [x*[y*(0..z))] where \( x \) is the number of screen pixels in the horizontal dimension, \( y \) is the number in the vertical dimension, and \( z \) is the number of pixel values. The screen pictures.

\textit{pre: char} \rightarrow \text{char.} The predecessor function.

\textit{printer!}. An output channel.

\textit{random.} A dictionary containing the following three names.

\[
\begin{align*}
\text{init.} & \text{A program with one real parameter. Its execution assigns a value to a hidden variable.} \\
\text{next.} & \text{A program. Its execution assigns the next value in a random sequence to the hidden variable.} \\
\text{value: real} & \rightarrow \text{real} \rightarrow \text{real.} \text{A reasonably uniform function of the hidden variable over the interval between the arguments.}
\end{align*}
\]

\textit{real.} The real numbers.

\textit{realtext: real} \rightarrow \text{text} A textual representation of a real number. MAYBE \text{erealrtext} and \text{frealtext}

\textit{return: char}. The return or newline character.
round: real → int. \( r - 0.5 \leq \text{round} r < r + 0.5 \)
screen!. An output channel.
session: text. A text expression giving all keystrokes on channel key since the start of a session.
sign: real → (−1, 0, 1).
sort: *ord → *ord where ord = real, char, [*ord].
stop :: wait (∞; ∞; ∞; ∞; ∞; ∞)
subst: all → all → * all → * all → all. subst \( x \) \( y \) \( s \) is a string formed from \( s \) by replacing all occurrences of \( y \) with \( x \). Substitute \( x \) for \( y \) in \( s \).
suc: char → char. The successor function.
tab: char.
text = *char.
textreal: text → real. If the argument represents a real number, the result is the represented number.
textcom: text → com. If the argument represents a complex number, the result is the represented number.
time!. An input channel that gives the time as a string of six numbers. For example, 1947; 9; 16; 14; 24; 30.095 is the time 1947 September 16 at 2:24 pm and 30.095 seconds.
timetext: (5*nat; real) → text. A readable form of the time. For example, \( \text{timetext} (1947; 9; 16; 14; 24; 30.095) = \text{“1947 Sep 16 at 2:24 pm”} \)
trig. A dictionary containing the following names. (The domains and ranges should probably include complex numbers, but I have forgotten my complex analysis.)
arccos: \( \$\langle r: \text{real} \rightarrow -1 \leq r \leq +1 \rangle \rightarrow \$\langle r: \text{real} \rightarrow 0 < r < \pi/2 \rangle \). An approximation to a trigonometric function.
arcsin: \( \$\langle r: \text{real} \rightarrow -1 \leq r \leq +1 \rangle \rightarrow \$\langle r: \text{real} \rightarrow 0 < r < \pi/2 \rangle \). An approximation to a trigonometric function.
arctan: \( \text{real} \rightarrow \$\langle r: \text{real} \rightarrow 0 < r < \pi/2 \rangle \). An approximation to a trigonometric function.
cos: \( \text{real} \rightarrow \$\langle r: \text{real} \rightarrow -1 \leq r \leq +1 \rangle \). An approximation to a trigonometric function.
sin: \( \text{real} \rightarrow \$\langle r: \text{real} \rightarrow -1 \leq r \leq +1 \rangle \). An approximation to a trigonometric function.
tan: \( \$\langle r: \text{real} \rightarrow \exists i: \text{int} \cdot (r = (2 \times i + 1) \times \pi) \rangle \rightarrow \text{real} \). An approximation to a trigonometric function.
trim: text → text. A text formed from the argument by removing all leading and trailing \( \text{space} \), \( \text{tab} \), and \( \text{return} \) characters.
true: bin. screen! true prints true
wait. A program with one parameter of type \( 5*\text{nat}; \text{real} \). Its execution does nothing but take the length of time given by the argument. See \( \text{await} \) and \( \text{time} \).

Grammar LL\((1/2)\)

In this grammar, for each nonterminal, every production except possibly the last begins with a different terminal. So director sets are not needed, and that's why I call it LL\((1/2)\).
name phraseaftername

programafterphrase . phrase programafterphrase
|| phrase programafterphrase
empty
name identifier modifier
modifier _ identifier modifier
empty
phraseafternewname : data
= data
:: phrase
!
== name
open name
empty
phraseaftername := data
! data
? data
:: phrase
argument
data comparand aftercomparand
comparand element afterelement
element item afteritem
item term afterterm
term factor afterfactor
factor # factor
- factor
~ factor
+ factor
~~ factor
\sqrt{ factor
\box{ factor
* factor
primary factorafterprimary
primary number
primary text
if data then data else data fi

do program result data od
{ data }
[ data ]
( data )
\langle name : primary \rightarrow data \rangle
name
argument number argument
primary text argument
if data then data else data fi argument
do program result data od argument
{ data } argument
[ data ] argument
( data ) argument
\langle name : primary \rightarrow data \rangle argument
For efficiency, the productions (except possibly the last) for each nonterminal should be placed in order of frequency. The following nonterminals can be eliminated: program name data comparand element item term.

**Grammar LR(1/2)**

The following grammar has no reduce-reduce choices and no shift-reduce choices. It has shift-shift choices. Such a grammar is commonly called LR(0), but it shouldn't be called that because we do
need to look at 1 symbol of input to choose among shift actions, so I'll compromise and call it $LR(^{1/2})$.

```
program
  phrase
  program . phrase
  program || phrase

phrase
  new name : data
  new name = data
  new name :: phrase
  new name !
  new name => data
  new name
  old name
  open name
  new open name
  close name
  name := data
  name ! data
  name ? data
  name !
  name ?
  name :: phrase
if data then program else program fi
for name := primary do program od
do program od

procedure
  ( program )
  procedure argument
name

data
  data = comparand
  data + comparand
  data < comparand
  data > comparand
  data ≤ comparand
  data ≥ comparand
  comparand
comparand
  comparand , element
  comparand , ... element
  comparand | element
  comparand < data > element
  element
element
  element ; item
  element ; ... item
  element ` item
  item
item
  item + term
  item – term
  item + term
```
item ∪ term

term
term × factor
term / factor
term ∧ factor
term ∨ factor
term ∆ factor
term ∇ factor
term ∩ factor

factor
+ factor
– factor
# factor
~ factor
√ factor
□ factor
* factor
primary * factor
primary → factor
primary ↑ factor
primary ↓ factor
primary

primary
primary argument
primary @ argument
argument

argument
number
text
[ data ]
{ data }
( data )
⟨ name : primary → data ⟩
if data then data else data fi
do program result data od

name
identifier
name _ identifier

Problems and Concerns

According to the grammars, we can't have ⟨x: 0..5 → 6⟩ and we can't have ⟨x: 0..5 → ok⟩ and we can't have for x:= 0;..5 do ok od. We need parentheses around 0..5 and 0;..5. It would be nice to not need the parentheses, but I don't see how.

Do we need var and chan parameters?

To execute a program stored on someone else's computer, I just invoke that remote program. For efficiency, it might be best to compile that remote program for my own computer and run it locally.
The question is how to get that program to read from my local keyboard and write to my local screen. We could add

```
! data
? data
```

with no channel name to mean the local screen and keyboard, but what about the local cursor and speakers? Should we treat channel names dynamically? Do `chan` parameters solve the problem?

When we use `eval` and `exec`, what do names mean? In `eval` “`x`” and `exec` “`x:= x+1`”, what `x` are we talking about? I think the use of `eval` and `exec` is equivalent to allowing programs to be assignable objects.

The following is not a semantic problem, but it is an implementation problem.

```
screen! 3
```

should print `3`, and similarly

```
screen! 1/0
```

should print `1/0`

```
screen! [0; 1; 2] 3
```

should print `[0; 1; 2] 3`

```
screen! (r: rat → 5) (1/0)
```

should print `5`

```
screen! 1/0 = 1/0
```

should print `true`

```
screen! [0; 1; 2] 3 = [0; 1; 2] 3
```

should print `true`

The syntaxes

```
new name :: program
name :: program
do name :: program od
```

do not adequately show how much program is being named. The syntaxes

```
new name :: do program od
name :: do program od
do name :: program od
```

each do half the job.

Should programmed data be

```
do program result data od
```

or

```
do program . data od
```

or

```
do program . data od
```

We can name (label) a (piece of) program in order to define it where it is used, as in

```
loop:: if x=0 then ok else x:= x–1. loop fi
```

Shouldn't we be able to do the same for data? For example,

```
1 + (fact=:: (n: nat → if n=0 then 1 else n×fact(n–1) fi)) 9 × 2
```

Same as

```
do new fact = (n: nat → if n=0 then 1 else n×fact(n–1) fi) result 1 + fact 9 × 2 od
```

Should it be

```
name =: data
```

or

```
name = data
```

hard for LL and LR

```
( name =: data )
```

hard for LL

```
( name = data )
```

hard for LL and LR

Using `=` would require the name to be fresh, and would reduce error detection.

It seems partly redundant to have both dictionaries and documents with links. Maybe a document can serve as a dictionary: clicking a link to another document is opening, and going back is closing.
But dictionaries can be opened and closed by a program, whereas clicking can be done only by a person. Clicking displays a document; opening a dictionary doesn't.

Should there be interactive variables? The syntactic cost is one keyword and one production.

```
new interactive name : data
```

Is there a shorter keyword? Or, perhaps the cost is one symbol and one production.

```
new name =: data
```

Should there be variable definition with initialization?
```
new name : data := data
```

Should there be a one-tailed if?
```
if data then program fi
```

Should we allow $A := 5$ as an abbreviation of $A := 3 \rightarrow 5 | A$?

Should there be variable definitions and functions and procedures with grouped variables?
```
new x, y, z: int
\langle a, b, c: \text{nat} \rightarrow a+b+c \rangle
\langle a, b, c: \text{nat} \rightarrow \text{if } a=0 \text{ then } x:= b \text{ else } x:= c \text{ fi} \rangle
```