A demonstration of the `program` environment

Martin Ward
martin@gkc.org.uk

June 5, 2013

Contents

1 Example with first_set and first_set.

The `program` style defines two environments, `program` and `programbox` for typesetting programs and algorithms. Within the program environment:

1. Newlines are significant;
2. Each line is in math mode, so for example spaces in the input file are not significant;
3. The command \\ within a line causes an extra linebreak in the output;
4. The indentation of each line is calculated automatically;
5. To cause extra indentation, use the commands `\tab` to set a new tab, and `\untab` to remove it (see the examples below);
6. Vertical bars are used to delimit long variable names with underscores (and other unusual characters).

```
testing | in verbatim

testing | and @ in verbatim
```

Here is a small program: $\mathsf{first\_set} := \{\, x \mid x^2 + y_1 > 0 \,\}$ It shows how to typeset mathematics as part of a program. Since each line is typeset in maths mode, all spacing is done automatically. The set brackets expand automatically, for example in this program (which also demonstrates the `\tab` and `\untab` commands):

$t := \left\{ x \mid \dfrac{x}{y} = z \right\};$
$t := t \setminus u;$
$z := a + b + c + d$
$\quad + e + f + g$
$\quad + h + i + j;$
if $x = 0$ **then** $y := 0$ **fi**

You can use variable_names in text or math mode: variable_name$^2 = 2$. Names can have odd_characters:!@#$%^&*:;_like_this!.

Note that `\(` and `\)` are redefined to typeset a program in a minipage. (This is useful in running text, or to keep a short program all on one page). There is some notation for sequences: $\langle x_1, x_2, \ldots, x_n \rangle$ and for universal and existential quantifiers: $\forall x.\, \exists y.\, y > x$ (yes, I use these in my programs!)

I often use bold letters to represent program fragments, formulas etc. so I have set up commands **S**, **R** etc. for the most common ones. The commands have one argument (a subscript, eg $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{S}_{23}$) or a sequence of "prime" characters: $\mathbf{S}'$, $\mathbf{S}''''$ etc. If you want both a subscript and one or more primes, then you must use maths mode, eg $\mathbf{S}'_2$ Consider the difference between typing ``` ``\S2'' ``` which gives "$\mathbf{S}_2$" and ``` ``$\S2''$'' ``` which gives "$\mathbf{S}''_2$". Outside maths mode, `\S` assumes any primes after a subscript are either closing quotes or apostrophes.

Here are two program examples with different indentation styles. Note that all indentation is calculated automatically in either style:

if $\mathbf{T}_1$
then if $\mathbf{T}_2$
then if $\mathbf{T}_3$
then $\mathbf{S}_4$
else $\mathbf{S}_3$
fi
else $\mathbf{S}_2$
fi
else $\mathbf{S}_1$
fi;

if $\mathbf{T}_1$ **then if** $\mathbf{T}_2$ **then if** $\mathbf{T}_3$ **then** $\mathbf{S}_4$
else $\mathbf{S}_3$ **fi**
else $\mathbf{S}_2$ **fi**
else $\mathbf{S}_1$ **fi**;

Note that **then** and **else** should be at the *start* of a line (as in the examples above), not at the end. This is so that you can line them up in short **if** statements, for example:

if $x = 1$ **then** a_long_procedure_name(arg1, arg2, ...)
else another_long_procedure_name(arg1, arg2, ...) **fi**

If the test is long, then you probably want an extra linebreak:

```
if a_long_boolean_function_name?(arg1, arg2, ... )
  then a_long_procedure_name(arg1, arg2, ... )
   else another_long_procedure_name(arg1, arg2, ... ) fi
```

Compare this with the following (which has linebreaks in the "wrong" places):

```
if a_long_boolean_function_name?(arg1, arg2, ... ) then
                                        a_long_procedure_name(arg1, arg2, ... )
                                    else
                                        another_long_procedure_name(arg1, arg2, ... ) fi
```

Just to show that | still works normally to indicate the placing of vertical lines) in the preamble of a tabular (or array) environment:

Statement	Conditions
$\mathbf{S}_1$	$\mathbf{B}_1$
$\mathbf{S}_2$	$\mathbf{B}_2$

2 Procedures and Functions

Turning on line numbering here. Also using the algoritm enviroment to number the algorithms within the sections.

Algorithm 2.1

```
(1)  A fast exponentiation function:
(2)  begin for i := 1 to 10 step 1 do
(3)              print(expt(2, i));
(4)              newline() od
(5)  where
(6)  funct expt(x, n) ≡
(7)    ⌈z := 1;
(8)     while n ≠ 0 do
(9)             while even(n) do
(10)                    n := n/2; x := x * x od;
(11)            n := n − 1; z := z * x od;
(12)    z⌋.
(13) end
```

First line is line **??**, last is line **??**. Line **??** is what makes this function fast!

Algorithm 2.2

```
(1)  A fast exponentiation procedure:
(2)  begin for i := 1 to 10 step 1 do
```

```
(3)              expt(2, i);
(4)              newline() od        This text will be set flush to the right margin
(5)  where
(6)  proc expt(x, n) ≡
(7)     z := 1;
(8)     do if n = 0 then exit fi;
(9)        do if odd(n) then exit fi;
(10)          comment: This is a comment statement;
(11)          n := n/2; x := x * x od;
(12)       {n > 0};
(13)       n := n − 1; z := z * x od;
(14)    print(z).
(15) end
```

An action system equivalent to a **while** loop:

```
(1) actions A :                    ≈    (1) while B do S od
(2) A ≡ if B then S; call A
(3)              else call Z fi.
(4) endactions
```

Note the use of `\(` and `\)` to enclose the two program boxes. Turning off line numbers here.

Dijkstra conditionals and loops:

if $x = 1 \to y := y + 1$
$\Box\ x = 2 \to y := y^2$
$\ldots$
$\Box\ x = n \to y := \sum_{i=1}^{n} y_i$ **fi**

do $2|x\ \wedge\ x > 0 \to x := x/2$
$\ \Box\ \neg 2|x \to x := |x + 3|$ **od**

Loops with multiple **exit**s:

```
do do if B1 then exit fi;
       S1;
       if B2 then exit(2) fi od;
   if B1 then exit fi od
```

I hope you get the idea!

3 A Reverse Engineering Example

Here's the original program:

Algorithm 3.1

```
var ⟨m := 0, p := 0, last := " "⟩;
  actions prog :
  prog ≡
    ⟨line := " ", m := 0, i := 1⟩;
    call inhere.
  l ≡
    i := i + 1;
    if (i = (n + 1)) then call alldone fi;
    m := 1;
    if item[i] ≠ last
      then write(line); line := " "; m := 0;
           call inhere fi;
    call more.
  inhere ≡
    p := number[i]; line := item[i];
    line := line ++ " " ++ p;
    call more.
  more ≡
    if (m = 1) then p := number[i];
                    line := line ++ ", " ++ p fi;
    last := item[i];
    call l.
  alldone ≡
    write(line); call Z. endactions end
```

And here's the transformed and corrected version:

Algorithm 3.2

```
⟨line := " ", i := 1⟩;
while i ≠ n + 1 do
      line := item[i] ++ " " ++ number[i];
      i := i + 1;
      while i ≠ n + 1 ∧ item[i] = item[i − 1] do
            line := line ++ ", " ++ number[i]);
            i := i + 1 od;
      write(line) od
```

Below are the same programs in a bold serif style with underlined keywords, using the command `\bfvariables`:

```
var ⟨m := 0, p := 0, last := " "⟩;
  actions prog :
  prog ≡
    ⟨line := " ", m := 0, i := 1⟩;
```

```
    call inhere.
l  ≡
    i := i + 1;
    if (i = (n + 1)) then call alldone fi;
    m := 1;
    if item[i] ≠ last
      then write(line); line := " "; m := 0;
           call inhere fi;
    call more.
inhere  ≡
    p := number[i]; line := item[i];
    line := line ++ " " ++ p;
    call more.
more  ≡
    if (m = 1) then p := number[i];
                    line := line ++ ", " ++ p fi;
    last := item[i];
    call l.
alldone  ≡
    write(line); call Z. endactions end
```

```
⟨line := " ", i := 1⟩;
while i ≠ n + 1 do
      line := item[i] ++ " " ++ number[i];
      i := i + 1;
      while i ≠ n + 1 ∧ item[i] = item[i − 1] do
            line := line ++ ", " ++ number[i]);
            i := i + 1 od;
      write(line) od
```

In my opinion, the `\sfvariables` style looks much better. The `\bfvariables` style was the default, but this was changed with version 3.3.11.