Tutorial for TUF-PL

Rainer Glaschick, Paderborn
2020-01-27

Contents

1. Introduction
1.1. Hello, World!
1.2. Print table of squares: loops
1.3. Command line parameters
1.4. Print squares again
1.5. Square root
1.6. Rational Numbers
1.7. Scan Functions

1. Introduction

This is a short tutorial for the programming language TUF-PL; for details, see the language specification and the documentation for the standard library functions.

TUF-PL is a precompiler for the C programming language, thus it is fairly easy to port the compiler — written in TUF-PL — to most computers.

The differences and highlights are:

- Block structuring by indentation, like in PYTHON; line oriented
- No keywords in the core language, only symbols and bigraphs
- Function definition and calls by word sequences with interspersed parameters, often no parenthesis needed
- Integer and string addressed dynamic rows and arrays
- Arbitrary precision integer numbers and rational numbers
- dynamic UTF strings
- fields for extra data in rows and arrays
- attributes for information about an item (object)
- fault items for reliable error handling

The example programs may be tested online at http://rclab.de/TUF-Online, but note that the examples are all indented to mark them as code; to run the program, you have to undo this indentation[1].

The term *bigraph* is used for two special characters without white space in between.

All programs need to include the definitions of the standard library, as shown in the first example. For easier reading, it will be omitted from the other examples.

1.1. Hello, World!

Here is the inevitable program that just prints a message:

```
\ the famous sample program prints just a greeting
main (parms):+
  print "Hello, world!"

\+ TUFPL_stdlib.tufh
```

producing the output

```
Hello, world!
```

First of all, the backslash exclusively starts comments and pragmas (that control the compiler); in our example, they start in column 1 (remember that the examples are indented for readability).

As in C, the main program is called main with the parameter array `parms`, that will be explained later. A function definition like `main (parms)` always starts in column 1 and ends in a colon; here with the bigraph `:+` to make it externally visible.

The main program has only one statement, that must be indented, i.e. not start in column 1 of a line. Here, it is a call to the standard library function `print` with one parameter, the string `"Hello, world!"`.

The missing parenthesis around the parameter are correct; you will soon observe — and hopefully appreciate — the few situations in which balancing of parenthesis is needed.

The last line includes from a file the templates (i.e. definitions) of the library functions used, `print ()` in this case. You may select a different library, e.g. one using French:

```
\ célèbre programme pour émettre un salut
main (paramètres):+
  tirer "Bonjour, le monde!"
\+ TUFPL_stdlib.fr.tufh
```

Please supply one in Russian, Italian, Spanish or whatever you like.

TUF-PL is fully UTF-aware, and it's easy, as there are no keywords; just translate the function templates.

1.2. Print table of squares: loops

As the very first computers were made to compute, the could not print character strings, just numbers. Thus, printing a table of squares is one of the earliest programming tasks:

```
\( squares: Print squares of numbers
\)
main (parms):+
  i =: 1
  ?* i < 11
     print i __ i*i
     i =+ 1
```

The output should be:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

First, the block comment is a bit unusual, using `/(` to open and `/(` to close; but as said before, a backslash is only used for comments, and everything that starts with a backslash is a comment or a pragma[2].

The first line is an assignment that puts the integer number `1` into the variable `i`; but — as PASCAL and ALGOL users might appreciate, and C and JAVA programmers might find at least funny — the assignment symbol is not just the equals sign, but the bigraph `=:` (two special characters glued)[3]. In contrast to ALGOL and PASCAL, the symbol starts with an equals sign, because there are more assignment symbols like `=+` and `=-`.

The next line is a simple loop that repeats while the expression `i<11` is true. The body of the loop is again indented, but still only one line, the `print` function with a string to print.

To compose the print line, the underscore is used to combine strings. While in general numbers and strings may not be used interchangeably, the `print` function as well as the catenation operator denoted by the underscore _ use a built-in conversion function to convert any number to a string in a standard format.

Thus, the print function still sees only one parameter, and this is the string expression `i __ i*i`. (The double underscore inserts a blank, while the single underscore glues two strings tightly together).

Finally, we have to increment the number in `i`. For this, the increment variant of an assignment is used, which is actually a shortcut for

```
i =: i + 1
```

Of course, any expression that provides an integer value is possible. Note again the inversion over the increment in C, in order that all assignments start with an equals sign. Moreover, the amount to be added is after the plus character.

There is no such thing as the ++ operator in C and JAVA. While I loved it for the compact and tricky programming it made possible when I learned C 30 years ago, it makes programs harder to understand. At that time the performance gains might have been valuable, but the risk of unreliable and buggy programs is much larger. As early programming instructions wrote: Use parenthesis; the compiler knows the rules exactly, but you as a human will sometimes err.

Of course, the normal way to write the body is:

```
?# i =: from 1 upto 10
  print i __ i*i
```

using the scan function `from () upto ()` to provide integers as a number generator. Scan functions are simple yet powerful and can generate all kinds of items; this will be explained later.

1.3. Command line parameters

You might already know that old-fashioned programmers like using a command line interface that allows to provide parameters.

The following program prints these parameters:

```
\ printparms: Just echo the parameters
main (parms):+
  ?# i =: from 1 upto parms.last
    print i __ parms[i]
  print "Total: " _ i - 1
```

If run with the parameters `A bc 9`, the result should be:

```
1 A
2 bc
3 9
Total: 3
```

As there are many parameters possible, the command line shell creates a list of them, which the TUF-PL runtime converts to a table (a *row*), which is the parameter of `main()`.

Again a scan function is used to deliver integer numbers, this time upto the value of the expression `parms.last`, which is a property (*attribute*) of `parms`, in this case the largest index.

After the printed list, the total number of lines before is printed; at this scan provides the next (out of bounds) integer on termination, we have to subtract 1; parenthesis may be used to show this:

```
print "Total: " _ (i - 1)
```

1.4. Print squares again

In the next example, the squares are calculated without multiplications, and the number is given as a parameter:

```
main (parms):+
  max =: integer from string parms[1] else 10
  \ squares by x² = (x-1)² + 2x - 1
  x =: 0
  sum =: 0
  ?# i =: from 1 upto max
    x =+ i + i - 1
    print i __ x
    sum =+ x
  print "sum=" _ sum
```

giving

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
sum=385
```

The maximum number to be squared is given as a parameter on the command line; this is a string which must be converted to a (binary) integer number.

So its time to explain now the probably most intriguing feature of TUF-PL: function calls do not need parenthesis. [4]

Each function is a template of words and parameters, separated by blanks; the template for the above function written as:

```
integer from string (str) else (default):
```

Looking for a function call, the words in the expression are compared to the function templates; and if one fits, the scan advances, which is the case for `integer`, `from` and `string`.

If there is no such word, as is the case for the 4th (`parms`), it is checked if there is a parameter symbol, which is a word in parenthesis in the template. If it is, an expression is parsed, which is in this case `parms[1]`, the first string on the command line. Then, matching resumes with the word `else`, which is a match. Finally, the expression `10` matches the second parameter in the template.

So, in C, the statement would be written as

```
max = integer_from_string_else(parms[1], 20);
```

and risk a compiler warning because `max` may already be a function.

Note that as TUF-PL does not use keywords, you are free to use `max`, `else`, etc, as variables.

It is good programming style in TUF-PL to use function templates with more than one word, which is required for functions without parameters. Because it is needed so often, `print` is one of those rare functions with one word only.

Although you may use `print` as a variable, this is also not good style:

```
print =: 7
print print
```

Variables need not and cannot be declared; the just come into existence by using the variable name; and are initialised to void. Misspelled variable names that are thus void can lead to subtle programming errors; but — with a bit of experience — the source is quickly identified, even if the compiler does not complain about variables that maybe used before initialised, i.e. are not target of an assignment before used in an expression.

There are no type declarations in TUF-PL, any variable can hold any kind of item, and may even change it:

```
x =: 'x'
x =_ 1     \ same as x =: x _ 1 and thus converts integer to string
! x = '1x'
```

1.5. Square root

A program to calculate a square root:

```
main(parms):+
  x =: float from string parms[1] else ()
  ? x = ()
    error print "Parameter " _ parms[1] _ " invalid, use decimal point: 2.0"
    :> 1
  print "sqrt(" _ x _ ")=" _ square root x

\( Iterative square root. If the initial value is > 1,
   then the values are monotonically falling, no epsilon needed
\)
square root (a):
  ! a >= 0.0                \ argument must be a positive rounded number
  ? a < 1.0                 \ iteration needs a > 1
    :> 1.0 / square root 1.0 / a
  x =: a
  ?*                        \ repeat until break
    y =: (x + (a/x)) / 2.0
    ? y >= x              \ if no longer falling values
      ?>                  \ break
    x =: y                \ save for comparison
  :> x
```

and, called with `2.0` as parameter, gives

```
sqrt(2)=1.41421
```

Besides numbers, strings and others, there is a special contents of a variable, called void, that is not a zero, not an empty string, neither false nor true, just nothing, indicated by a pair of parenthesis (the bigraph `()`).

This allows in the first line (of `main`) to indicate that the parameter string was not a valid number.

In the next line a plain condition — not a loop — is indicated by just a question mark, that questions the boolean expression that compares x to void. If the condition holds, processing continues with the indented block, giving a message to the error output, thus the function `error print ()` is used.

It follows a return with integer 1 as a return code from the whole program.

This time we use a function to do the calculation just to avoid to deeply indented code.

A function definition always starts in column 1, followed by the template, followed by a colon as last on the line. The function body is then just indented as usual.

The first statement is an assertion: an exclamation mark, followed by a boolean expression. The programmer wants to ensure that the parameter is not negative, and is a floating point number. If this is not the case, the program goes to an error stop. This is also the case if `x` is not a floating point number, as there are no automatic conversions between integer and floating point number and clearly not from strings.

The algorithm used is a bit unusual; you might miss an epsilon to terminate the iteration. This is annoying at least, as it requires to know the accuracy of the machine. But if started with a number greater 1.0, the values are monotonically falling in value, and then staying the same or oscillating. If there is no more progress, the condition `y >= x` holds, and the loop is terminated with the break symbol `?>`. The symbols for function return and break are intentionally similar.

When the loop terminates, the value in `x` is the root, and returned to the caller.

If a negative floating point number is supplied, there will be a failure message like:

```
Fatal: Assertion failed.
       at   12 in square root (a)
Called at    6 in main (parms)
```

telling that the co-operation between the function programmer and the main programmer was not close enough. I know of no type system that allows to declare a non-negative floating point number to catch such an error at compile time.

If you enter 0.0 at the command line, you would expect to crash with a division by zero in the line

```
:> 1.0 / square root 1.0 / a
```

But modern floating point number arithmetic results in `inf` for infinity, and its reciprocal is 0.0.

Instead of using the absolute value in the function, the input check is made better:

```
  x =: float from string parms[1] else ()
  ? x = () & x > 0
    error print ...
```

If you enter 0.0 at the command line, you would expect to crash with a division by zero in the line

```
:> 1.0 / square root 1.0 / a
```

But modern floating point number arithmetic results in `inf` for infinity, and its reciprocal is 0.0, so the correct result is given accidentally.

You might try to add the lines

```
? x = 0.0
    :> 0.0
```

unaware of the language description telling that equality comparisons for floating point numbers are dubious and thus not allowed; only order comparisons are sensible.

1.6. Rational Numbers

Integer numbers may be — as in PYTHON — arbitrary long and are not restricted to 64 bits; so there is not integer overflow. If accurate numbers are required that are not whole numbers, rational numbers are supported (currently only as 32 bit numbers, just to prove the concept):

```
\ sum up the fractions: 1 + 1/2 + 1/3 + 1/4 + ...
main (parms):
  x =: 0
  ?# i =: from 1 upto 10
    x =+ 1 / i
    print i __ x __ x.float   \ or ##x or float x
```

giving

```
1 1 1
2 3/2 1.5
3 11/6 1.83333
4 25/12 2.08333
5 137/60 2.28333
6 49/20 2.45
7 363/140 2.59286
8 761/280 2.71786
9 7129/2520 2.82897
10 7381/2520 2.92897
```

The division operator may be used either for pairs of floating point numbers or a pair of accurate numbers, which are integers and rationals (integers are rationals with denominator 1). As can be seen

from the 6th line of the output, common divisors in nominator and denominator are automatically cancelled (and if the denominator is 1, the result is given back as integer number).

The truncated division for integers is `//`, while the commonly used computer remainder is `%%`. See the documentation for reasons and variants.

1.7. Scan Functions

The already used scan function `from () upto ()` is written in TUF-PL:

```
from (start) upto (end):
  :> from start upto end step 1

from (start) upto (end) step (step):
  ? $# = ()               \ first call?
     $# =: start          \ yes, use start value
  |
     $# =+ step           \ no, increment
  rv =: $#                \ save
  ? $# > end | $# < start \ done ?
     $# =: ()             \ yes
  :> rv

main(parms):
  ?# i =: from 1 upto 6
      print i             \ prints 1 to 12
  print i                 \ prints 13
```

giving

```
1
2
3
4
5
6
7
```

The first two lines just map the absent upper limit to 1.

In the body for the next function, the special variable `$#` is used, called the *local context* of a scan. It is set to void at the begin of each scan. Upon entry to the function, this context variable is checked for void; if it is, its the first call, and the context variable set to it. Otherwise, it is incremented. Next, it is checked if the upper limit `last` has been exceed; if yes, the context variable is cleared to void. This signals the calling loop control that the loop is terminated, and the body of the loop, i.e. the print, is now skipped.

Because the check takes place after the assignment, the return value, which is the next value, is assigned to `i`. This is necessary in case the scans ends prematurely and returns a fault item to indicate this case.

Dropping the line `rv =: $#` and returning `$#` supplies void to `i` when the loop is ended.

Note that the scan can also be used to supply floating point or rational numbers:

```
main(parms):
  ?# i =: from 1 upto 2 step 1/10
    print i
```

gives

```
1
11/10
6/5
13/10
7/5
```

```
3/2
8/5
17/10
9/5
19/10
2
```

Scan functions look a bit clumsy because their initialisation take space, and the context variable `$#` looks ugly (it is choosen for similarity with `?#`, that supplies a number of values).

Surely scan functions can nested, e.g. to supply a multiplication table:

```
main (parms):
  ?# i =: from 1 upto 9
    ?# j =: from 1 upto 9
      print (pad left i*j to 3) nonl
    print ''
```

gives

```
1  2  3  4  5  6  7  8  9
2  4  6  8 10 12 14 16 18
3  6  9 12 15 18 21 24 27
4  8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

The print function `print () nonl` does not terminate with a line feed (the parenthesis are not necessary). Note that the pad function accepts either numbers or strings, and always delivers a string.

The tutorial currently ends here. If you would like to have it continued, send me your comments to `tufpl@rclab.de`.

---

[1]ok, I should use a better method for preparing this text...

[2]Future version might allow `)/` to close, and even `\*` and `*\`; but `/*` is an operator for the euclidean integer division in TUF-PL.

[3]Having spent many hours of debugging by accidentally writing C assignments in expressions where equality comparison was intended, and having started 50 years ago programming in ALGOL 60, I still do not like the equation symbol for assignments.

[4]I learned this from the STAGE2 macroprocessor published in 1970 by W. M. Waite: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.965&rep=rep1&type=pdf.