



txpp: the Tech-X Pre-Processor

Peter Messmer and The VORPAL Team
Tech-X Corporation

VORPAL Users Group Meeting, Nov. 11, 2007

- **Anatomy of VORPAL input files**
 - For details, see Chet Nieter's introduction
- **'Standard' use of txpp**
 - Command line arguments
 - Simple substitutions
- **Macros**

Anatomy of a VORPAL input file



- **Input file sets up the simulation**
- **Is organized as a hierarchy of blocks**
 - Between opening and closing tag:
`<Tag blockname>`
`</Tag>`
- **Blocks contain**
 - Parameters
 - Comments
 - txpp directives
 - Other blocks
- **Outermost block is implicit**
 - No need to specify it

txpp offers a mechanism to parameterize input files



- Behavior of pre-processor is controlled by *directives*
 - Lines with '\$' as first non-blank character
 - Allows for indentation
- Common use: Symbols
 - *Formulas*: assign value to symbol
 - Every occurrence of a symbol is substituted by the value

```
$ dx = 10.                # assign a constant
$ dy = dx / 2.           # assign computed value
$ c  = 2.9979e8          # short symbol names are ok
$ dt = 1/(c*sqrt(1/dx**2 + 1/dy**2))
```

```
numPhysCells = [N N]
lengths = [L $dy * N$]
```

```
dt = dt
```

Invocation of txpp



- **Standard use: Translate mysim.pre into a VORPAL input file mysim.in**

```
txpp.py --prefile=mysim.pre      traditional
txpp.py mysim.pre                short form
```

- **Useful command line options**

```
--help      -h      Display list of options
--stdout    -s      Send output to stdout
--output=FILE -o FILE Chose a different output file name
--quiet     -q      Do not include directives in output
--dump_symbol_tables -d Include top-level symbol table

[ --ndim=N      Assign N to symbol NDIM ]
SYMBOL=VALUE   Assign VALUE to SYMBOL
```

- **Command line defined symbols**

- Any top-level symbol can be assigned a value
- Override assignment in input file
- Are not evaluated until used (true for any symbol)

More txpp directives



- **Import**

```
$ import fragments.pre           # reuse trusted code. Imports  
                                # can be arbitrarily nested
```

- **Debugging**

```
$ print hello world! `DX`=       # print message to stdout  
$ exit                           # terminate translation  
$ requires a, b                   # stop translation if symbols a or b is not  
                                # defined
```

- **Conditional translation, optional else clause**

```
$ if NDIM > 3  
$   print This does not make sense!  
$   exit  
$ else  
$   print moving ahead  
$ endif
```

- **Repeated translation**

```
$ i = 0  
$ while i < 3  
$   print i  
$     lowerBounds = [ $i * dx$ ]  
$   i = i + 1  
$ endwhile
```

That's it...



- .. what a VORPAL user should need to know
- **Goal of txpp: support libraries, simplify complex inputfiles**
 - Macros!

- **Reusable blocks of text**

- Substitution occurs once macros are used

```
<macro mymacro>
    macros are helpful
</macro>
mymacro                -> macros are helpful
$ helpful = cool!
mymacro                -> macros are cool!
```

- **Can span multiple lines**

```
<macro laser>
yeeEmField
    <BoundaryCondition leftBC>
        kind = variable
        lowerBounds = [-1 -1 -1]
        upperBounds = [1 NY NZ]
        omega = 1e12
    </BoundaryCondition>
</macro>
```

```
<EmField myfiled>
    kind = laser
</EmField>
```

txpp translation steps



- **Concatenate lines**
- **Tokenize**
 - python lexical analysis
 - removes all white spaces, matched quotes
- **For all tokens in a line (after an eventual '=')**
 - While token is found in symbol table:
 - Tokenize value
 - Insert value token list into token list
 - Evaluate token as python expression
 - If error, substitute sting value
- **Concatenate result**
 - reintroduce white spaces
- **For formulas: evaluate result python expression**
 - If error, use string value

Macros can take parameter



```
<macro laser(freq)>
yeeEmField
  <BoundaryCondition leftBC>
    $ requires NY, NZ           # avoid long parameter lists
    $ omega = freq * 2 * pi    # blocks open new scope
    upperBound = [1 NX NZ]
    omega = omega
  </BoundaryCondition>
</macro>

<EmField myLaser>
  kind = laser(3e9)
</EmField>
```

Macros can take parameter



```
<macro laser(freq, misc)>
yeeEmField
  <BoundaryCondition leftBC>
    $ requires NY, NZ # avoid long parameter lists
    $ omega = freq * 2 * pi # blocks open new scope
    upperBound = [1 NX NZ]
    omega = omega
  </BoundaryCondition>
  misc
</macro>

<macro myBC(name)> # create some parameterized BC
  <BoundaryCondition name>
    ..
  </BoundaryCondition>
</EmField>

<EmField myLaser> # create a laser with the new BC
  kind = laser(3e9, myBC( rightBC ))
</EmField>
```

Macros can be overloaded



Macro can be overloaded. Particular instance is identified by number of parameters.

```
<macro laser(freq, misc)> # same as in previous slide
```

```
...
```

```
</macro>
```

```
<macro laser(freq)> # overloaded macro for simpler invocation
```

```
laser(freq, '')
```

```
</macro>
```

```
<EmField myLaser> # use first instance of macro
```

```
kind = laser(3e9, myBC( rightBC ))
```

```
</EmField>
```

```
<EmField myLaser> # use simplified instance
```

```
kind = laser(3e9)
```

```
</EmField>
```

Functions: special macros



- **Parameters are substituted**

```
<macro prod(a, b)>  
a*b  
</macro>
```

```
prod(3, 5)           -> 3 + 5  
prod(3+5, 7)        -> 3+5*7           -> unexpected behavior!  
2*prod(3+2,1)       -> 2*3+2*1       -> unexpected behavior!
```

- **Function introduce the expected behavior**

```
<function prod(a, b)>  
a*b  
</function>
```

```
2*proc(3+2, 1)       -> 2 * ((3+2)*(1))
```

- **Particularly useful in expressions**
 - Can therefore be only one output line

Example: string concatenation



- Useful for parameterized input file (e.g. libraries)

```
<macro concat(a, b)>  
$ a = `a  
a b'  
</macro>
```

```
<Field concat(Field, X)>          -> <Field FieldX>  
<Field concat(Field, Y)42>       -> <Field FieldY42>
```

- **New pre-processor is backward compatible**
 - Old input files should still work
- **New pre-processor introduces macros**
 - Macros help to hide input file complexity
- **Goal is to reduce complexity of input files**
 - Library developers have to do a bit more work 😊