xudf

eXtended User Definable Format
version 0.1.0

Takashi Uneyama

Table of Contents

1 Introduction

The XUDF (eXtended User Definable Format) is an extension of the OCTA UDF (User Definable Format) for describing the input/output data for various simulators of computational physics. Although the philosophy of the OCTA UDF is interesting, but to read/write the OCTA UDF file, one needs the proprietary library. This avoids developments of free software / open source software. We hope the libraries and utilities to handle these format to be free. Besides, the official OCTA UDF handling library is not fast and difficult for users. The OCTA UDF syntax itself has some ambiguity and this confuses the users. The XUDF is written to be a free, open utility to handle the OCTA UDF files. You can convert OCTA UDF files into XUDF files, and then you can convert them into some script languages and handle it by ease.

Currently the XUDF provides the preprocessor for XUDF macro (and also for OCTA UDF) and the converters for script languages (Lua, Python, Ruby, Perl). You can also convert the OCTA UDF into the script languases directly, and then use it by your simulator.

NOTICE THAT THE XUDF IS NOTHING TO DO WITH OCTA NOR JAPAN RESEARCH INSTITUTE UNLIMITED. XUDF IS COMPLETELY UNOFFICIAL PACKAGE. ALSO NOTICE THAT THE XUDF IS DISTRIBUTED UNDER THE GNU GENERIC PUBLIC LICENSE WHICH IS NOT COMPATIBLE WITH THE LICENSE OF OCTA.

2 Install `xudf`

You can build ans install `xudf` from the source package. `xudf` is using GNU Autoconf/Automake, thus you can build and install it easily, just like the other free softwares.

First, extract the source package `xudf-0.1.0.tar.gz`. To extract it, do

```
$ zcat xudf-0.1.0.tar.gz | tar xvf -
```

or if you are using GNU tar, do

```
$ tar zxvf xudf-0.1.0.tar.gz
```

Now change the directory to the source tree and bild it.

```
$ cd xudf-0.1.0
$ ./configure
$ make
```

If the `make` is completed without errors, then install it.

```
$ su -
# make install
```

By default, `xudf` will be installed under `/usr/local`. For further information, invoke the following command.

```
$ ./configure --help
```

This will print the options for the `configure` script.

3 About XUDF (eXtended User Definable Format)

In this section, we show a simple XUDF files as examples. The XUDF files are similar to the OCTA UDF files since the XUDF is based on the OCTA UDF.

3.1 Primitive Data Types

The XUDF supports following primitive data types.

`int`	A 32-bit integer.
`long`	A 32-bit integer (this is the same as *int*).
`short`	A 16-bit integer.
`float`	A single-precision floating point real number.
`single`	A single-precision floating point number (this is the same as *float*).
`select`	A selectable, enumerated string (this is the same as *string*).
`double`	A double-precision floating point number.
`string`	A string, sequence of characters.
`KEY`	A string specifier for the variable (actually this is the same as *string*).
`ID`	A integer specifier for the variable (actually this is the same as *int*).

3.2 Definition and Value of Variables

First, we show the simplest XUDF file.

```
def {
    i: int
}

data {
    i: 123
}
```

This XUDF file means that we define the `int` type variable `i` and its value is `123`. The definition of the variables are in the `def` block;

```
def {
    name: type
    ...
}
```

where *name* is the name of the variable and *type* is the type of the variable. The value for the variable is set in the `data` block;

```
data {
    name: value
    ...
}
```

where *value* is data which is set to the variable *name*.

3.3 Arraies and Structures

More complex data structures, arraies and structures can be used in the XUDF.

```
def {
    a[]: int
    s: {
        b: double
        c: string
    }
}

data {
    a[]: [ 1 2 3 4 5 ]
    s: {
        1.234
        "abcde"
    }
}
```

In this case, `a` is an array of `int` and `s` is a structure of which members are `b` (`double` type) and `c` (`string` type). The array type variable can be defined by adding `[]` after its name (multidimensional array can be defined by adding `[]` twice or more). The structure type variable can be defined by combining variable definitions.

```
def {
    array_name[]: type
    structure_name: {
        name: type
        ...
    }
}
```

The data value definitions for an array and a structure is as follows.

```
data {
    array_name[]: [ data1, data2, data3, ... ]
    structure_name: { data1, data2, data3, ... }
}
```

The array type or the structure type can be nested. The following is a bit complicated but collect XUDF.

```
def {
    a[]: {
        b: {
            c: int
            d[]: double
        }
        e[][]: int
    }
}
```

```
data {
    a[]: [
        { {
            1
            [ 1.0 2.0 3.0 ]
            }
            [ [ 1.0 2.0 ]
              [ 3.0 4.0 ] ]
        }
        { {
            2
            [ 4.0 5.0 6.0 ]
            }
           [ [ 5.0 6.0 ]
             [ 7.0 8.0 ]]
        }
    ]
}
```

3.4 Class Definitions

The `class` definition is useful if there are many structures of which member is the same.

```
def {
    class vector: {
        x: double
        y: double
        z: double
    }

    r: vector
    v: vector
}

data {
    r: { 1.0 2.0 3.0 }
    v: { 4.0 5.0 6.0 }
}
```

In this XUDF file, we define a new `class` named `vector`. The calss definition is just like the structure variable definition except for that there are class specifier `class`. The defined `class` can be used as the variable type, like the other primitive variable types (`int, double, ...`).

```
def {
    class class_name: {
        name: type
        ...
```

```
    }
}
```

3.5 Select Definitions

The *select* type variable is enumerated string type variable.

```
def {
    b: select {
        "true"
        "false"
    }
}

data{
    b: "true"
}
```

In this example XUDF, `b` is a `select` variable which can take the value `"true"` or `"false"`. The select variable can be treated as a variant of string variable.

The definition and set of `select` variables is as follows.

```
def {
    name: select {
        "item1"
        "item2"
        ...
    }
}

data {
    name: "defined_item"
}
```

where "*defined_item*" is one of "*item1*", "*item2*", ...

3.6 Headers

We can store the data or the information about the simulation in the special block, the `header` block.

```
header {
    def {
        EngineType: string
        EngineVersion: string
        IOType: string
        ProjectName: string
        Comment: string
    }
```

```
    data {
        EngineType : "my simulator"
        EngineVersion : "0.1.0"
        IOType : "in"
        ProjectName : "my project"
        Comment : "an input file for my simulator 0.1.0"
    }
}
```

We put the `def` block and the `data` block in the `header` block. The variables which can be defined in the `def` block in the `header` block is as follows.

`EngineType`	(*string*) The type (or name) of the simulation engine which uses the file.
`EngineVersion`	(*string*) The versioin of the simulation engine.
`IOType`	(*string*) The input/output type for the file ("in" for input, "out" for output, "in/out" for both input and output).
`ProjectName`	(*string*) The name of the project which developped the simulation engine.
`Comment`	(*string*) The comment about the file.

3.7 Records

Sometimes we need to use sequential data sets or similar data sets. The `record` block can be used for such a purpose.

```
def {
    n: int
    t: double
}

record "record 0" {
    data {
        n: 0
        t: 0.0
    }
}

record "record 1" {
    data {
```

```
            n: 1
            t: 0.5
        }
    }

    record "record 2" {
        data {
            n: 2
            t: 1.0
        }
    }
```

In this XUDF, we have three `record` blocks. Each `record` block has the `data` block. The value in the `data` block in the `record` block refers the same variable, but they are isolated and distinguished by the record identifier.

The record definition is as follows.

```
    record "record_identifier" {
        data {
            name: value
            ...
        }
    }
```

where "*record_identifier*" is an identifier which distinguishes records.

4 The XUDF Preprocessor (`xudfpp`)

4.1 Overview of the XUDF preprocessor (`xudfpp`)

The XUDF preprocessor, `xudfpp` is the preprocessor for the OCTA UDF and the XUDF files. It processes the backslashed command (for example, `\begin{def} ... \end{def}`) into the plain XUDF format (in this case, `def { ... }`). It also remove the comments (`/* ... */` and `// ...`) and processes some ambiguous / inappropriate OCTA UDF syntax into the strict XUDF sytax. It is useful to convert the OCTA UDF files into the XUDF files, or use some macro expansions for the XUDF.

In most cases, the users do not need to invoke `xudfpp` directly. Just like the C preprocessor `cpp`, `xudfpp` will be called from the other programs, automatically. The OCTA UDF to Perl / Python / Ruby / Lua converters (`udf2pl, udf2py, udf2rb, udf2lua`) call `xudfpp` automatically.

4.2 Introduction to (`xudfpp`)

Currently, `xudfpp` supports only some simple macro expansions. The most useful expansion is one for `\begin{...} ... \end{...}` block normally used in the OCTA UDF files (the OCTA UDF uses these syntax to define blocks such as the data block or the definition block). `xudfpp` processed `\begin{...} ... \end{...}` blocks into the XUDF blocks. For example, the OCTA UDF file

```
//  OCTA UDF input

\begin{def}
    a: double
    b[]: int
\end{def}

\begin{data}
    a: 1.23
    b[]: [4, 5, 6]
\end{data}
```

will be converted to the following XUDF file.

```
def {
a : double
b [ ] : int
}

data {
a : 1.23
b [ ] : [ 4 5 6 ]
}
```

As expressed, several parts are processed. First, the comment at the first line is removed. Second, the `\begin{...} ... \end{...}` block is converted into the XUDF block syntax, Third, the commas in array data (`b`) is removed (because the XUDF do not allow commas there).

Next we show the useful macro expansion. You can define constant value as a new macro command, using `\definie{...}{...}` syntax. The first argument for `\define` corresponds to the new macro identifier and the second argument corresponds to its value. The simple example is as follows.

```
\define{pi}{3.141592}

def {
    a: double
    b: {
        c: double
        d: double
    }
}

data {
    a: \pi
    b: {
        1.234
        \pi
    }
}
```

The new macro command, `\pi`, will be expanded into `3.141592` defined in the first line.

```
def {
a : double
b : {
c : double
d : double
}
}

data {
a : 3.141592
b : {
1.234
3.141592
}
}
```

4.3 Invoking `xudfpp`

The format for running the `xudfpp` program is:

```
$ xudfpp [input] [output] option ...
```

The input file (*input*) and the output file (*output*) can be specified. If both files are specified, `xudfpp` uses them as the input/output files. If only one file (*input*) is specified, `xudfpp` takes it as the input file and print the processed data into the standard output stream. If no file is specified, `xudfpp` reads data from the standard input stream and then process it and print into the standard output stream.

`xudfpp` supports the following options:

`-I directory`
: Add the directory *directory* to the head of the list of directories to be searched for header files. If you use more than one `-I` option, the directories are scanned in left-to-right order.

`-D name` Define the name *name* as a macro with null string.

`-D name=definition`
: Define the name *name* as a macro with definition *definition*. Currently, `xudfpp` does NOT permit the override of macro definitions, and once the macro with a name *name* is defined, it cannot be redefined or removed.

`--help`
`-h` Show summary of options.

`--version`
`-v` Show version of program.

4.4 Macro Processing by `xudfpp`

4.4.1 `\begin{...}` and `\end{...}` (blocks)

The OCTA UDF type block, begins with `\begin{...}` and ends with `\end{...}` is processed into the XUDF block.

The `\begin` part,

```
\begin{type}
```

is processed into

```
type {
```

unless the *type* is `record` or `global_def`. For the `record` block,

```
\begin{record}{"record_identifier"}
```

is processed into the following form.

```
record "record_identifier" {
```

For the `global_def` block,

```
\begin{global_def}
```

is processed into the following form.

```
global {
```

This is because the XUDF do not support `global_def` block (instead, the `global` block is used).

The `\end` part,

```
\end{type}
```

is simply replaced by

```
}
```

4.4.2 `\include{...}` (inclusion of file)

The `\include` syntax, sometimes used in the OCTA UDF, is processed as follows. The `\include` syntax,

```
\include{"file_to_include"}
```

is removed by the `xudfpp`. `xudfpp` then opens the file named *file_to_include* and processes it. If the processing of the opend file (*file_to_include*) is completed (if the `xudfpp` reaches the end of the file), `xudfpp` continues to process the original file.

4.4.3 `\define{...}{...}` (macro definition)

The `\define` syntax, which is newly introduced to the `xudfpp`, defines new macros.

```
\define{identifier}{value}
```

defines new macro command, `\identifier`. the new command *identifier* is expanded into the value, *value*. The `\define` syntax itself is removed by the `xudfpp`.

The new macro command can be used wherever, once it is defined.

```
\identifier
```

is expanded into

```
value
```

Note that the redefinition and the removal of the macro is not supported currently. Once the macro is defined, all the following new macro definition is expanded.

4.4.4 `<KEY>` and `<ID>` (variable types)

The OCTA UDF supports two special variable types, `<KEY>` and `<ID>` (though they are not often used). The XUDF does not support such a syntax, thus the `xudfpp` removes `<` and `>`. That is, the variable type definition including `<` and `>`

```
<type>
```

is processed into the following form.

```
type
```

4.4.5 , (commas)

In the OCTA UDF, commas are allowed to be used as the separator for data. The XUDF do not support these ambiguous syntax and `xudfpp` removes these commas. The array definition with commas

```
[ element1, element2, element3, element4, ... ]
```

is processed into the array definition which do not contains commas.

```
[ element1 element2 element3 element4 ... ]
```

4.4.6 `/* ... */` and `// ...` (comments)

The C style comments and the C++ style comments are removed by the `xudfpp`.

The C comment begins with `/*` and ends with `*/`.

```
/* comments ...
...
...
... */
```

The `xudfpp` removes it.

The C++ comments begnis with `//` and ends with the end of the line.

```
// comments ...
```

The `xudfpp` removes it, too.

5 The XUDF to Script Language Converter (`xud2pl,xudf2py,xudf2rb,xudf2lua`)

5.1 Overview of the XUDF to Script Language Converter (`xudf2pl,xudf2py,xudf2rb,xudf2lua`)

The XUDF to script language converter (`xudf2pl,xudf2py,xudf2rb,xudf2lua`) converts the XUDF files into some popular script language files. The XUDF files cannot be handled directly from programs, and the converter is useful to use the XUDF files as the input of the programs.

Currently there are 4 languages supported by the XUDF – Perl, Python, Ruby and Lua. The XUDF files processed by the converter have the data as the hash table (this may be called as dictionary or associated list, in some languages).

It is noted that the XUDF to script language converter does NOT use the `xudfpp` automatically. If you want the data processed by the `xudfpp` automatically, use the UDF to script language converter (`udf2pl, udf2py, udf2rb, udf2lua`) instead. It preprocesses and converts the input file into the script language file.

5.2 Introduction to the XUDF to Script Language Converter

The 4 lauguages (Perl, Python, Ruby and Lua) is supported currently. In this section, we show the conversion of the simple XUDF file into the Perl script. The following is the example input XUDF file.

```
def {
    r: {
        x: double
        y: double
        z: double
    }
    a: {
        b: int
        c[]: double
    }
}

data {
    r: {
        1.0
        2.0
        3.0
    }
    a: {
        1.23
        [ 45 67 89 ]
```

```
        }
    }
```

If we convert it by using `xudf2pl`, we get the following perl script as output.

```
# generated by xudf2pl

%record = ();

%data = (
"r" => { "x" => 1.0, "y" => 2.0, "z" => 3.0, },
"a" => { "b" => 1.23, "c" => [ 45, 67, 89, ],
},
);
```

Here the variable `%data` is the hash table which contains all the data in the input XUDF file. It has the same structure as the input XUDF file. For example, The variables `r` and `a` in the input XUDF is the structure type, and they are the structured variable (strictly speaking, it is the hash table) in the output Perl script. The array variable `c` is also expressed as the array in the output Perl script.

To access these variables from Perl, first evaluate the output Perl script. Then these variables are stored in the memory and are accessible from Perl. For example, the following script print the variables `r.x` and `a.c[0]` in the output Perl script (`output.pl`)

```
#!/usr/bin/perl

require 'output.pl';

print $data{"r"}{"x"},"\n";
print $data{"a"}{"c"}[0],"\n";
```

If you use the other converters (`xudf2py,xudf2rb,xudf2lua`), the resulting output is each script language, and it can be handled just like the case of Perl.

For Python, the output script converted by `xudf2py` is

```
# generated by xudf2py

record = {}

data = {
'r' : { 'x' : 1.0, 'y' : 2.0, 'z' : 3.0, },
'a' : { 'b' : 1.23, 'c' : [ 45, 67, 89, ],
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/python

from output import *

print data["r"]["x"]
print data["a"]["c"][0]
```

For Ruby, the output script converted by `xudf2rb` is

```
# generated by xudf2rb

record = {}

data = {
"r" => { "x" => 1.0, "y" => 2.0, "z" => 3.0, },
"a" => { "b" => 1.23, "c" => [ 45, 67, 89, ],
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/ruby

load "output.rb"

print $data["r"]["x"]
print $data["a"]["c"][0]
```

For Lua, the output script converted by `xudf2lua` is

```
-- generated by xudf2lua

record = {}

data = {
r = { x = 1.0, y = 2.0, z = 3.0, },
a = { b = 1.23, c = { 45, 67, 89, },
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/lua

loadfile("output.lua")()

print(data.r.x)
print(data.a.c[1])
```

Note that the array index of Lua starts from 1, not 0.

5.3 Invoking `xudf2pl,xudf2py,xudf2rb,xudf2lua`

The format for running the `xudf2pl,xudf2py,xudf2rb,xudf2lua` program is:

```
$ xudf2pl [input] [output] option ...
$ xudf2py [input] [output] option ...
$ xudf2rb [input] [output] option ...
$ xudf2lua [input] [output] option ...
```

The input file (*input*) and the output file (*output*) can be specified. If both files are specified, the converter uses them as the input/output files. If only one file (*input*) is specified, the converter takes it as the input file and print the processed data into the standard output stream. If no file is specifled, the converter reads data from the standard input stream and then process it and print into the standard output stream.

`xudf2pl,xudf2py,xudf2rb,xudf2lua` supports the following options:

`--help` `-h`	Show summary of options.
`--version` `-v`	Show version of program.

6 The UDF to Script Language Converter (`ud2pl,udf2py,udf2rb,udf2lua`)

6.1 Overview of the UDF to Script Language Converter (`udf2pl,udf2py,udf2rb,udf2lua`)

The UDF to script language converter (`udf2pl,udf2py,udf2rb,udf2lua`) converts the OCTA UDF files into script language files. It is the wrapper script which processes the input file by using `xudfpp` and then convert it with `xudf2pl,xudf2py,xudf2rb,xudf2lua`.

It is convenient to use the converter to read your OCTA UDF files from your programs or process the data in the OCTA UDF files by script languages.

6.2 Introduction to the UDF to Script Language Converter

The 4 lauguages (Perl, Python, Ruby and Lua) is supported currently (this is the same as the XUDF to script language converter). In this section, we show the conversion of the simple UDF file into the Perl script, as shown in the XUDF converter case. The following is the example input OCTA UDF file.

```
\begin{def}
    r: {
        x: double
        y: double
        z: double
    }
    a: {
        b: int
        c[]: double
    }
\end{def}

\begin{data}
    r: {
        1.0,
        2.0,
        3.0
    }
    a: {
        1.23,
        [ 45, 67, 89 ]
    }
\end{data}
```

This OCTA UDF file has the same data as the XUDF file shown in the XUDF section. If we convert it by using `xudf2pl`, we get the following perl script as output.

```
# generated by xudf2pl
```

```
%record = ();

%data = (
"r" => { "x" => 1.0, "y" => 2.0, "z" => 3.0, },
"a" => { "b" => 1.23, "c" => [ 45, 67, 89, ],
},
);
```

Here the variable `%data` is the hash table which contains all the data in the input XUDF file. It has the same structure as the input XUDF file. For example, The variables `r` and `a` in the input XUDF is the structure type, and they are the structured variable (strictly speaking, it is the hash table) in the output Perl script. The array variable `c` is also expressed as the array in the output Perl script.

To access these variables from Perl, first evaluate the output Perl script. Then these variables are stored in the memory and are accessible from Perl. For example, the following script print the variables `r.x` and `a.c[0]` in the output Perl script (`output.pl`)

```
#!/usr/bin/perl

require 'output.pl';

print $data{"r"}{"x"},"\n";
print $data{"a"}{"c"}[0],"\n";
```

If you use the other converters (`xudf2py`,`xudf2rb`,`xudf2lua`), the resulting output is each script language, and it can be handled just like the case of Perl.

For Python, the output script converted by `xudf2py` is

```
# generated by xudf2py

record = {}

data = {
'r' : { 'x' : 1.0, 'y' : 2.0, 'z' : 3.0, },
'a' : { 'b' : 1.23, 'c' : [ 45, 67, 89, ],
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/python

from output import *

print data["r"]["x"]
print data["a"]["c"][0]
```

For Ruby, the output script converted by `xudf2rb` is

```
# generated by xudf2rb

record = {}
```

```
data = {
"r" => { "x" => 1.0, "y" => 2.0, "z" => 3.0, },
"a" => { "b" => 1.23, "c" => [ 45, 67, 89, ],
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/ruby

load "output.rb"

print $data["r"]["x"]
print $data["a"]["c"][0]
```

For Lua, the output script converted by `xudf2lua` is

```
-- generated by xudf2lua

record = {}

data = {
r = { x = 1.0, y = 2.0, z = 3.0, },
a = { b = 1.23, c = { 45, 67, 89, },
},
}
```

and the script which print the variables `r.x` and `a.c[0]` is as follows.

```
#!/usr/bin/lua

loadfile("output.lua")()

print(data.r.x)
print(data.a.c[1])
```

Note that the array index of Lua starts from 1, not 0.

The UDF to script language converter can handle the XUDF files as input, too. This is because the converter uses `xudfpp` to process the input file. We get the XUDF files by processing the OCTA UDF files as well as the XUDF files.

6.3 Invoking `udf2pl,udf2py,udf2rb,udf2lua`

The format for running the `udf2pl,udf2py,udf2rb,udf2lua` program is:

```
$ udf2pl [input] [output] option ...
$ udf2py [input] [output] option ...
$ udf2rb [input] [output] option ...
$ udf2lua [input] [output] option ...
```

The input file (*input*) and the output file (*output*) can be specified. If both files are specified, the converter uses them as the input/output files. If only one file (*input*) is specified, the converter takes it as the input file and print the processed data into the standard output stream. If no file is specified, the converter reads data from the standard input stream and then process it and print into the standard output stream.

`udf2pl,udf2py,udf2rb,udf2lua` supports the following options:

`-I` *`directory`*
: Add the directory *directory* to the head of the list of directories to be searched for header files. If you use more than one `-I` option, the directories are scanned in left-to-right order. This option is passed to `xudfpp`.

`-D` *`name`*
: Define the name *name* as a macro with null string. This option is passed to `xudfpp`.

`-D` *`name=definition`*
: Define the name *name* as a macro with definition *definition*. Currently, `xudfpp` does NOT permit the override of macro definitions, and once the macro with a name *name* is defined, it cannot be redefined or removed. This option is passed to `xudfpp`.

`--help`
`-h`
: Show summary of options.

`--version`
`-v`
: Show version of program.

7 Complete Syntax for XUDF

The syntax for the XUDF in the extended Backus-Naur format (EBNF) is shown in this section. Note that this syntax may contain bug, because currently the XUDF parser is written by hand without `yacc` and thus actually the following EBNF is not directly used.

```
xudf-file = { block }


block = def-block
      | global-block
      | data-block
      | header-block
      | unit-block
      | record-block


header-block = ‘header’ ‘{’ { header-sub-block } ‘}’
header-subblock = def-block
                | data-block


record-block = ‘record’ ‘[’ string ‘]’ ‘{’ { data-block } ‘}’


def-block = ‘def’ ‘{’ { type-definition [ type-unit-definition ] } ‘}’
type-definition = primitive-definition
                | struct-definition
                | class-definition
type-unit-definition = ‘[’ specifier ‘]’
primitive-definition = variable-name ‘:’ specifier
variable-name = specifier { ‘[’‘]’ }
struct-definition = variable-name ‘:’ ‘{’ { struct-element-defintion } ‘}’
struct-element-definition = primitive-definition
                          | struct-definition
class-definition = ‘class’ specifier ‘:’ ‘{’ { struct-element-defintion } ‘}’


global-block = ‘global’ ‘{’ { type-definition [ type-unit-definition] } ‘}’


data-block = ‘data’ ‘{’ data-definition ‘}’
data-definition = { specifier ’:’ data-value }
data-value = number
           | string
           | struct-value
           | array-value
struct-value = ‘{’ { data-value } ‘}’
```

```
array-value = ‘[’ { data-value } ‘]’


unit-block = ‘data’ ‘{’ { unit-definition } ‘}’
unit-definition = unit-constant-definition
                | unit-unit-definition
unit-constant-definition = specifier ‘=’ number
unit-unit-definition = ‘[’ specifier ‘]’ ‘=’ [ number ] ‘[’ unit-complex ‘]’
unit-complex = unit
             | number
             | ‘(’ unit-complex ‘)’
             | unit-complex unit-operation unit-complex
             | unit-complex ’^’ number
unit-operation = ‘+’
               | ‘-’
               | ‘*’
               | ‘/’


letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’
       | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’
       | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’
       | ‘y’ | ‘z’
       | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’
       | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’
       | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’
       | ‘Y’ | ‘Z’
digit = ‘0’ | nonzero-digit
nonzero-digit = ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’
              | ‘9’

sign = ‘+’
     | ‘-’
exponent = ’e’
         | ’E’
integer = [ sign ] nonzero-digit { digit }
float = [ sign ] nonzero-digit { digit } ’.’ { digit } [ sign exponent { digit } ]
      | [ sign ] ’0’ ’.’ { digit } [ sign expontent { digit } ]
number = integer
       | float

specifier = specifier-starter { specifier-continuer }
specifier-starter = letter
                  | ’_’
specifier-continuer = specifier-starter
                    | digit
```

```
character = letter
          | digit
          | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘!’ | ‘#’ | ‘$’ | ‘%’
          | ‘&’ | ‘’’ | ‘(’ | ‘)’ | ‘{’ | ‘}’ | ‘[’ | ‘]’
          | ‘~’ | ‘^’ | ‘‘’ | ‘@’ | ‘;’ | ‘:’ | ‘<’ | ‘>’
          | ‘,’ | ‘.’ | ‘_’ | ‘?’
escape-sequence = ’\’ character
                | ’\’ ’\’
                | ’\’ ’"’
string = ‘"’ { string-character } ‘"’
string-character = character
                 | escape-sequence
```

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you

indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright
interest in the program ‘Gnomovision’
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Concept Index