

# **PPI Ruleset Language and Examples**

Todd King  
Institute of Geophysics and Planetary Physics  
UCLA

Applications and ruleset language designed by: Steve Joy, Todd King, Joe Mafi  
and Erin Means

Last Updated: July 14, 2006

Available from: <http://www.igpp.ucla.edu/pds/ruleset>

Introduction.....	1
Ruleset Language.....	1
Comments .....	1
Variables .....	2
Directives .....	3
ABORT .....	3
COPY .....	3
ELSE .....	3
ELSEIF .....	4
GLOBAL .....	4
IF .....	4
/IF .....	5
IGNORE .....	5
INCLUDE .....	5
MESSAGE.....	6
OPTION .....	6
OUTPUT .....	7
RUN .....	7
TEMPLATE.....	7
Plug-ins .....	8
Application: PPIRuleset.....	9
Examples.....	9

## Introduction

The Ruleset description maker is based on the ruleset processing engine and ruleset language designed by Steve Joy, Todd King, Joe Mafi and Erin Means at the Institute of Geophysics and Planetary Physics at UCLA. The ruleset engine takes as input a ruleset specified in the ruleset language (described later) and a template for a description. A template description can be any text file including XML and PDS Labels. PDS labels are syntactically validated during processing, whereas other formats are not. A template contains variables that will be replaced by the ruleset engine. The ruleset details how to define the variables needed by the template. Variables can be defined by direct assignments, through conditional branching, using plug-ins (external applications that return rulesets) or including other rulesets. The output file is generated after the ruleset is processed and all variables are replaced in the template. This is depicted in Figure 1

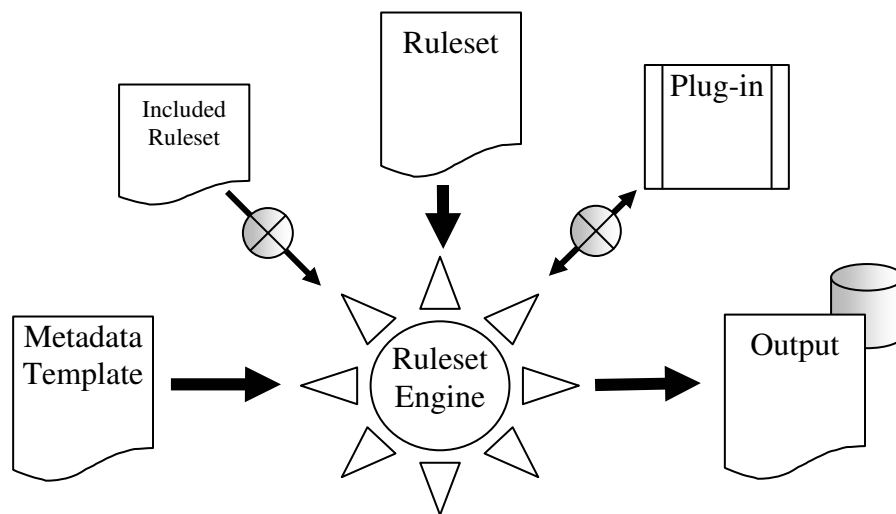


Figure 1: Basic functions of ruleset processing.

## Ruleset Language

The ruleset scripting language is designed to offer a simple and extensible framework for setting the values of variables and to use those variables in creating formatted output. There are three distinct entities of a rule set: comments, variables and directives.

### Comments

A comment is any line of text that begins with either “#” or a “/”. All of the following are permissible comments:

```
# This is a comment using shell script commenting
// This is a comment using C++ style commenting
/* This is a comment using C style commenting */
```

It is important to note that while you can create comments which look like C style comments, the begin (/\*) and end (\*/) comment rules do not apply. Therefore, you can not create comment blocks which span multiple lines or embed comments within a line.

## Variables

The central entity of a ruleset is the variable. A variable is simply a named value. The syntax for defining a variable is:

```
$name = value
```

where the “\$” is literal, *name* is the tag for the variable and *value* is the text to associate with the name. A name may contain any alphanumeric character and the underscore (\_). A value may contain any text. A value may span multiple lines if it is enclosed in quotes (“”) or is a list enclosed in curly braces (i.e. { }) or parenthesis (i.e. ( )).

Some example variable statements are:

```
$DATA_TYPE = document
$DESCRIPTION = "This is a lot of text which
                can span multiple lines."
$LIST = {      "Value1",
               "Value2",
               "Value3" }
```

There are several variables that are defined by the ruleset processor when a ruleset is used with a file. These variables are:

FILE_PATH	The path portion of the file specification.
FILE_NAME	The name of the file currently being processed. This includes the file extension, but does not include any path information.
PATH_NAME	The combined path and filename.
FILE_EXT	The portion of the file name that follows the last period (.). The extension of the file name.
FILE_BASE	The portion of the file name that excludes the file extension.
FILE_SIZE	The size in bytes of the file.
FILE_DATE	The date portion of the file creation timestamp. It is in PDS style (yyyy-MM-dd)
FILE_TIME	The complete timestamp of the file in PDS style (yyyy-MM-ddThh:mm:ss)

When a variable is used the order in which the variable is resolved is by first looking for the variable in the ordinary list of variables, then looking in the list of global variables, then the list of system defined variables. These means that is possible to override a system variable with a local or global variable definition.

## Directives

Directives are commands to the ruleset processor which control which rules are executed and provide an interface to external files or applications (plug-ins) for acquiring rulesets. A directive may have one or more arguments. A directive and any related arguments are enclosed in angle brackets (<>). The canonical form of a directive is:

```
<directive [argument ...]>
```

The available directives are:

### ABORT

The ABORT directive ends the processing of the rules and reports that all processing should end. The rule form using the ABORT directive is:

```
<ABORT>
```

An example is:

```
<ABORT>
```

A typical use of ABORT is when some error is encountered and no recovery is possible.

### COPY

The COPY directive instructs the ruleset processor to copy a file from one location to another. The rule form using the INCLUDE directive is:

```
<COPY source destination>
```

The *source* can be a relative or absolute path. If it is a relative path it will be relative to the file currently being processed. The *destination* can be a directory or a filename. If it is a directory the will be file copied to the *destination* using its original file name. If the destination includes a file name, then the file will be copied to a file with the name given. If destination is omitted, then the current value of FILE\_PATH is used.

An example is:

```
<COPY /stuff/mag.fmt new.fmt>
```

will load the file “mag.fmt” from the “/stuff” directory to the same directory as the file the ruleset is running on. The name of the copied file will be “new.fmt”.

### ELSE

The ELSE directive marks the beginning of a block of rules which will be executed if the condition of the preceding IF directive are not met. An ELSE block can only occur between an IF and /IF directive and marks the end of the preceding block of rules. The rule form for using the ELSE directive is:

<ELSE>

## ELSEIF

The ELSEIF directive marks the beginning of a block of rules which will be executed if the value associated with a variable matches the specified pattern. An ELSEIF block can only occur between an IF and /IF directive and marks the end of the IF block. An ELSEIF will be checked only if the proceeding IF or ELSEIF block did not match its pattern. The rule form using the ELSEIF directive is:

```
<ELSEIF variable = pattern>
```

An example is:

```
<IF $FILE_NAME = "*03">
# Do something
<ELSEIF $FILE_NAME = "*04">
# Do something else
</IF>
```

will be true if the value of \$FILE\_NAME ends with the characters "04".

## GLOBAL

The GLOBAL directive defines a variable that should persist between executions of individual rulesets. It is the responsibility of the calling application to preserve each global variable. The rule form using the GLOBAL directive is:

```
<GLOBAL variable value>
```

where *variable* is the name of the persistent variable and it is assigned *value*.

## IF

The IF directive marks the beginning of a block of rules which will be executed if the value associated with a variable matches the specified pattern. The rule form using the IF directive is:

```
<IF $variable = pattern>
or
<IF $variable != pattern>
or
<IF $variable>
```

where *variable* is the name of the variable and *pattern* is the pseudo regular expression of the pattern to compare to the value of *variable*. A pattern can contain any regular expression syntax with the following exception. All periods (.) are considered literal and

a star (\*) is considered a match for zero or more characters. If just the *\$variable* is specified then the existence of the variable is checked. If it is not defined *false* is returned. If it is defined, but is empty *false* is returned. Otherwise *true* is returned. Preceding the equal sign (=) with an exclamation point (!) means “not equal to”.

The end of the IF block of rules is marked with the </IF> directive.

An example is:

```
<IF $FILE_NAME = "*03">
# Do something
</IF>
```

will be true if the value of \$FILE\_NAME ends with the characters “03”.

## **/IF**

The /IF directive marks the end of the block of rules that was marked with the most recent IF directive.

```
<IF $FILE_NAME = "*03">
# Do something
</IF>
```

## **IGNORE**

The IGNORE directive ends the processing of the rules and reports that no output should be generated. The rule form using the IGNORE directive is:

```
<IGNORE>
```

An example is:

```
<IGNORE>
```

A typical use of IGNORE is when some condition in the ruleset is encountered that would make output undesirable. For example, when files with a particular extension are encountered and no label should be generated for such files.

## **INCLUDE**

The INCLUDE directive instructs the ruleset processor to open a file and load the contents as a set of rules. The rules will be run using the current variables and the resulting variables will be merged into the current list of variables. The rule form using the INCLUDE directive is:

```
<INCLUDE filename>
```

An example is:

```
<INCLUDE constant.rul>
```

will load the file “constant.rul” and interpret it as a set of rules.

## MESSAGE

The MESSAGE directive provides a means to display a message for the user. A message may span multiple lines. Each argument to the MESSAGE directive is displayed on a new line. Arguments may be quoted so that text that contains spaces can be displayed on a single line. Text may also contain references to variables which will be replaced before displaying the text. The rule form using the MESSAGE directive is:

```
<MESSAGE [arguments ...]>
```

An example is:

```
<MESSAGE “This is an example message.”>
```

will display the following:

```
This is an example message.
```

## OPTION

The OPTION directive sets the value of an option for the ruleset processor. Options are parameters that control how the various directives in the ruleset processor perform. The rule form using the OPTION directive is:

```
<OPTION option value>
```

The available options are:

PAD_FILE	Indicates whether to pad the output file. If <i>true</i> the file will be padded to the width set in PAD_WIDTH, otherwise the file will not be padded.
PAD_WIDTH	The width in characters to pad each line in a file.
INDENT	The number of spaces to pad the beginning of a line which has been wrapped
WRAP_LINE	The width in characters that each line will be wrapped.
FORCE_UPPER	Indicates that file names will be forced to uppercase. If <i>true</i> the file names will be converted to upper case, otherwise the filename will be unchanged.
EQUAL_AT	The position to align the equal sign following a keyword.



## OUTPUT

The OUTPUT directive defines the name of the file the output will be written. If the output file name is not specified it defaults to \$BASE\_NAME with the extension “.lbl”. The rule form using the OUTPUT directive is:

```
<OUTPUT filename>
```

An example is:

```
<OUTPUT example.lbl>
```

defines “example.lbl” as the name of the file to write the output.

## RUN

The RUN directive will execute a command, passing any number of arguments, and process the output from the command as a set of rules. A command may be an external application, script or executable process. In order to be used by a ruleset the output must be a set of rules which will be executed by the ruleset processor. Because of this requirement the command is also referred to as a “plug-in”. The rule form using the RUN directive is:

```
<RUN command [arguments...]>
```

Each argument passed to the command may contain references to variables. Each reference will be replaced with the current value of the variable prior to running the command. In this way, information defined in previous rules can be passed from plug-in to plug-in.

An example is:

```
<RUN extract $PATH_NAME>
```

will run the command “extract” which will presumably extract some information from a file.

## TEMPLATE

The TEMPLATE directive defines the file which will be used generating output. The template must be a PDS label. Each occurrence of a variable name in the template will be replaced with the value of the variable prior to generating an output file. The rule form using the TEMPLATE directive is:

```
<TEMPLATE filename>
```

An example is:

```
<TEMPLATE template.lbl>
```

defines “template.lbl” as the file which contains the template to use.

## Plug-ins

A plug-in is an external application, script or process which can be run from the ruleset processor. A plug-in can be passed any number of arguments on its command line and the output is collected by the ruleset processor and interpreted as a set of rules. Suppose you want to write an application which will scan a data file and extract the start and stop times from the file. You could have an application called “extract” which accepts one command line argument, the name of the file to scan. So, the rule you would use to call the plug-in is:

```
<run extract $PATH_NAME>
```

The output of “extract” should be something like:

```
$START=2003-04-23T00:00:00  
$STOP=2003-04-24T00:00:00
```

which will define the variables START and STOP. These variables could then be used in a template with something like this:

```
START_TIME = $START  
STOP_TIME = $STOP
```

A plug-in can return any legal ruleset and may include IF/ELSEIF/ELSE branches, INCLUDE, RUN rules.

## Application: PPIRuleset

In the java implementation of the ruleset processor the class "pds.ruleset.PPIRuleset" is executable and provides all the necessary services to use the Ruleset language. The command line arguments to "pds.ruleset.PPIRuleset" are the name of the file that contains the ruleset and the file to apply the ruleset to. It can be run with the command:

```
java pds.ruleset.PPIRuleset example.rul datafile
```

Complete documentation for the Java ruleset processor (pds.java) is located at:  
<http://www.igpp.ucla.edu/pds/ruleset/doc>

## Examples

Here is a complete example which demonstrates a large part of the capabilities of the ruleset processor. Suppose we have a set of constants that are the same for every label we want to create. We also have an application called "extract" which will determine the start and stop times for a data file. Then the following ruleset will load the constants, load a template, run the plug-in and generate an output label paired with the data file.

example.rul

```
<INCLUDE constant.rul>
<TEMPLATE template.lbl>
<OUTPUT $BASE_NAME.lbl>
<RUN extract $PATH_NAME>
```

The contents of the constant.rul file is:

constant.rul

```
$PDS_VERSION = PDS3
$DSID = DSID_1_0
$STD_PROD_ID = DATA
$PROD_TYPE = DATA
$REC_TYPE = FIXED
$RECL = 120
$RECS = 512
$START_SCLK = 2400:0
$STOP_SCLK = 2500:0
$HOST_NAME = Galileo
$HOST_ID = GLL
$ORBIT = 1024
$TARGET_LIST = JUPITER
$INST_NAME = MAG
$INST_ID = MAG
$STD_PROD_DESCR = "This is a short description"
$FF_ABSTRACT = "This is a much longer
multi-line type description which
spans multiple lines."
$INTERCHANGE = ASCII
$RECS = 10
$COLS = 4
$RECL = 64
$FMT = Unknown
$COL_DESCR = "What?"
$HDR_BYTES = 80
$HDR_TPYE = FIXED
$HDR_DESCR = "This is the header file"
```

The contents of the template file is:

template.lbl

```
PDS_VERSION_ID      = $PDS_VERSION
DATA_SET_ID         = "$DSID"
STANDARD_DATA_PRODUCT_ID = "$STD_PROD_ID"
PRODUCT_ID          = "$FILE_BASE"
PRODUCT_TYPE        = "$PROD_TYPE"
PRODUCT_CREATION_TIME = $FILE_TIME

RECORD_TYPE         = $REC_TYPE
RECORD_BYTES        = $RECL
FILE_RECORDS        = $RECS

START_TIME          = $START_TIME
STOP_TIME           = $STOP_TIME
SPACECRAFT_CLOCK_START_COUNT = "$START_SCLK"
SPACECRAFT_CLOCK_STOP_COUNT = "$STOP_SCLK"

INSTRUMENT_HOST_NAME = "$HOST_NAME"
INSTRUMENT_HOST_ID   = "$HOST_ID"
ORBIT_NUMBER         = $ORBIT
TARGET_NAME          = $TARGET_LIST
INSTRUMENT_NAME       = "$INST_NAME"
INSTRUMENT_ID        = "$INST_ID"
DESCRIPTION           = "
$STD_PROD_DESCR"

NOTE                = "
$FF_ABSTRACT"

^TABLE              = "$FILE_BASE.FFD"
OBJECT              = TABLE
  INTERCHANGE_FORMAT = "$INTERCHANGE"
  ROWS               = $RECS
  COLUMNS            = $COLS
  ROW_BYTES          = $RECL
  ^STRUCTURE         = "$FMT"
  DESCRIPTION        = "
    $COL_DESCR"
END_OBJECT          = TABLE

^HEADER             = "$FILE_BASE.FFH"
OBJECT              = HEADER
  BYTES              = $HDR_BYTES
  HEADER_TYPE        = "$HDR_TPYE"
  DESCRIPTION        = "$HDR_DESCR"
END_OBJECT          = HEADER
END
```

The output from the “extract” command will look like:

#### Outut from extract

```
$START_TIME = 2002-10-6T00:00:00  
$STOP_TIME = 2003-01-12T00:00:00  
$START_SCLK = 2400:0  
$STOP_SCLK = 2500:0
```

If the execution of this ruleset runs without returning an IGNORE then a label file will be written which has the same base name as the file that is processed and will look like:

output.lbl

PDS\_VERSION\_ID = PDS3  
DATA\_SET\_ID = "DSID\_1\_0"  
STANDARD\_DATA\_PRODUCT\_ID = "DATA"  
PRODUCT\_ID = ""  
PRODUCT\_TYPE = "DATA"  
PRODUCT\_CREATION\_TIME = 2003-11-17T10:30:00.000

RECORD\_TYPE = FIXED  
RECORD\_BYTES = 120  
FILE\_RECORDS = 512

START\_TIME = 2002-10-6T00:00:00  
STOP\_TIME = 2003-01-12T00:00:00  
SPACECRAFT\_CLOCK\_START\_COUNT = "2400:0"  
SPACECRAFT\_CLOCK\_STOP\_COUNT = "2500:0"

INSTRUMENT\_HOST\_NAME = "Galileo"  
INSTRUMENT\_HOST\_ID = "GLL"  
ORBIT\_NUMBER = 1024  
TARGET\_NAME = Jupiter  
INSTRUMENT\_NAME = "MAG"  
INSTRUMENT\_ID = "MAG"  
DESCRIPTION = "  
This is a short description "

NOTE = "  
This is a much longer  
multi-line type description which  
spans multiple lines ."

^TABLE = "EXAMPLE.FFD"  
OBJECT = TABLE  
INTERCHANGE\_FORMAT = "ASCII"  
ROWS = 10  
COLUMNS = 4  
ROW\_BYTES = 64  
^STRUCTURE = "Unknown"  
DESCRIPTION = "  
What?"  
END\_OBJECT = TABLE

^HEADER = "EXAMPLE.FFH"  
OBJECT = HEADER  
BYTES = 80  
HEADER\_TYPE = "FIXED"  
DESCRIPTION = "This is the header file"  
END\_OBJECT = HEADER  
END