1 Products and Parameters .

A product (e.g.: `:exe`) is a Unix file, a directory or a list of files that can be requested from Eli. Parameters (e.g.: `+fold`) allow the requestor to control some characteristics of the requested product.

Generated files can be displayed by appending `>`, file-lists can be viewed with `:viewlist` and generated directories by listing their contents with `!ls`. Files from a generated directories can be selected by appending `/file` to the request.

For further details see Eli Products and Parameters.

1.1 Processor Generation

`:exe` Executable file containing the generated processor.

`:source` File-List with all source and include files making up the processor.

`:allspecs`
File-List with all files defining a processor.

`:gencode` File-List with all files generated by Eli from your specifications.

`:fwGen` Directory with all files specified in one .fw file.

`:ligaResults`
File-List with all files generated by Liga from your specifications.

`+define` cpp directive for C compilation.

`+fold` To suppress case distinctions in identifiers and keywords.

`+ignore` To switch off the verification of the presence of certain include files.

`+parser` Selects the parser generator:pgs/cola.

1.2 Generating Specifications

`:bnf` File containing your concrete grammar in BNF notation.

`:consyntax`
File containing complete parsing grammar.

`:abstree` File containing complete tree grammar.

Table of Contents

`:inst` File-List containing instantiated generic module.

`:kwd` Recognize specified literals as identifiers.

`+instance, +referto`
For instantiation of specification modules.

1.3 Diagnostics

`:warning` File containing Warnings noted while deriving a product.

`:error` File containing Errors noted while deriving a product.

`:warn, :err`
Unprocessed warning and error messages.

`:help` Executable for browsing Warning and error messages of a derivation. Messages contain references to documentation.

`:parsable`
File containing verification protocol of the parsability of the parsing grammar (LALR(1)).

`:showFe, :showMe`
File-List with 3 files containing information about the Lido specifications.

`:ExpInfo,:OrdInfo,:OptimInfo`
Files with Information from Liga on remote attribute access, attribute dependencies, attribute storage.

`:gorto` Start `gorto`, a graphical tool for attribute dependence analysis.

1.4 Testing a Generated Processor

`:stdout` Standard output from a test run, for example

```
input +cmd=(x.specs:exe):stdout
```

`:run` Execute the generated processor, for example

```
. +cmd=(x.specs:exe) input :run
```

`:output` Output files from a test run, for example

```
input +cmd=(x.specs:exe) :output !ls -l
```

`:profile` Execution profile.

`:dbx, :gdb`
Debug a program interactively at the source level.

`:bgdbx`, `:bggdb`
: Examine core dump from a test run.

`:mon` Monitor a program at the specification level.

`:mondbx`, `:mongdb`
: Monitor a program at the specification level.

`+arg` Command line arguments for processor execution (only usable with `:mon`)

`+debug` Flag to request debugging information in object files.

`+input` Directory containing files to be made available during execution.

`+monitor` Flag to request monitoring support.

`+printtokens`
: Flag to request that tokens be printed as they are read.

`+prof` Flag to request profiling code in object files.

`+stdin` File to be made available as standard input.

1.5 Producing Formatted Documents

`:dvi` DVI-file generated from a hypertext document.

`:fwTex` TeX file generated from a .fw file.

`:fwTexinfo`
: Hypertext document generated from a .fw file.

1.6 Information About the Derivation

`!:redo` Tell Eli to redo a derivation step, even though no inputs to it have changed.

`!:test` Ask Eli to check whether an object has been modified.

`!:inputs` A list of the objects on which this object directly depends.

`!:outputs`
: A list of the objects directly depending on this object.

2 Eli Specifications

The Eli user describes the subproblems of a particular text processing problem in files of different "type". The type is indicated by the file name extension. Any of these files can contain C-style comments and preprocessor directives such as `#include`, `#define` and `#ifdef`.

`.specs` A collection of subproblem descriptions, one per line:

```
word.gla
$/Tool/lib/Name/Nest.gnrc :inst
symbol.lido
```

`.gla` A description of the token structure of the input text:

```
ident : C_IDENTIFIER
string: $' (auxPascalString) [mkstr]
numb  : $[0-9] [mkint]
```

`.con` A description of the phrase structure of the input text:

```
def: set_name '=' '{' body '}' .
body: element+ .
cond : 'if' exp 'then' stmt $'else'.
```

`.lido` A description of the structure of a tree and the computations to be carried out on that tree:

```
ATTR Sym: int;
SYMBOL set_name INHERITS Entity END;
SYMBOL text COMPUTE
  PTGOut(
    PTGTable(
      CONSTITUENTS set_name.Sym
          WITH (int, ADD, ONE, ZERO)));
END;
RULE r_wall: wallspec ::= 'wall' pos ';'
COMPUTE
  wallspec.done = setwall(pos.x, pos.y);
END;
```

`.map` A description of the mapping between the parsing and the tree grammar.

`.ctl` Options for evaluator generation.

`.h, .c` C modules for user-supplied functions, variables, types etc.

`.head` Headers and macro definitions to be inserted into code generated from Lido:

```
#include "myproc.h"
#define MyValue(s) MyArray[s]
```

`.init, .finl`

C code to be executed before any processing begins (.init) or after all other processing is complete (.finl):

```
{ int s;
  s = GetValue(speed,1);
  setdelay(1000000/s);   }
```

.ptg A description of structured output text:

```
Seq: $ $
List: $ ",\n\t" $
```

.pdl A property definition language:

```
code : mytype; "kcode.h"
size : int;
```

.oil A description of operator overloading:

```
OPER iAdd(integer, integer): integer;
OPER rAdd(real, real): real;
INDICATION Plus: iAdd, rAdd, sUnion;
COERCION Float(integer): real;
```

.clp A description of command line arguments for the generated processor:

```
speed "-s" int
"-s determines steps per second";
```

.fw Combines a collection of strongly-coupled specifications with documentation describing their relationships:

```
@O@<c.ptg@>@{
Seq:   $ $
@}
@O@<c.lido@>@{
SYMBOL Entity INHERITS IdPtg END;
@}
```

.delit Specifies literals appearing in a type-‘con’ file that are to be recognized by special routines.

.gnrc Defines a generic specification module.

3 User Interface

Single characters are quoted with `+` in an Eli request; strings are quoted by enclosing them in apostrophes ('). Spaces and tabs are ignored, and `#` marks the rest of the line as a comment. The request `?` starts the documentation browser.

For further details see Interacting with Eli.

`object` Make a product up-to-date with respect to its inputs.

```
x.specs+monitor:exe # Make up-to-date
x.specs:parsable<   # To your editor
x.specs>            # To standard output
x.specs:exe>x.exe   # To file x.exe
x.specs:source>src  # To directory src
```

`!` Execute the remainder of the line as a shell command. If `!` is preceded by `object`, append the name of the up-to-date product to the end of the line.

`=` Query or set variables.

```
?=          # Show list of all variables.
Dir=?       # Show 'Dir' variable meaning.
History=    # Show the value of 'History'.
ErrLevel=1  # Set 'ErrLevel' to '1'.
```

`control character`

Request editing with history. Starred commands accept a repeat count (e.g. '+ESC 4 ^P+'). Arrow keys can be used to move in the history.

```
^A   Move to the beginning of the line

^B*  Move left in the line (left arrow)

^E   Move to the end of the line

^F*  Move right in the line(right arrow)

^N*  Next request in history (down arrow)

^P*  Previous request in history (up arrow)

^R*  Request a substring to search for
     String starts line if it begins with ^
     Search forward if repeat count given
```