OCaml Standard Tools

Standard Tools

.opt tools are the same tools, compiled in native-code, thus much faster.

ocamlopt[.opt]	native-code compiler
ocamlc[.opt]	bytecode compiler
ocaml	interactive bytecode toplevel
ocamllex[.opt]	lexer compiler
ocamlyacc	parser compiler
ocamldep[.opt]	dependency analyser
ocamldoc	documentation generator
ocamlrun	bytecode interpreter

Compiling

A unit interface must be compiled before its implementation. Here, ocamlopt can replace ocamlc anywhere to target asm.

ocamlc -c test.mli	compile an interface
ocamlc -c test.ml	compile an implementation
ocamlc -a -o lib.cma test.cmo	generate a library
ocamlc -o prog test.cmo	generate an executable
ocamlopt -shared -o p.cmxs test.cmx	generate a plugin

Generic Arguments

-config	print config and exit
-c	do not link
-o *target*	specify the target to generate
-a	build a library
-pp *prepro*	use a preprocessor (often camlp4)
-I *directory*	search directory for dependencies
-g	add debugging info
-annot	generate source navigation information
-i	print inferred interface
-thread	generate thread-aware code
-linkall	link even unused units
-nostdlib	do not use installation directory
-nopervasives	do not autoload Pervasives

Linking with C

-cc *gcc*	use as C compiler/linker
-cclib *option*	pass option to the C linker
-ccopt *option*	pass option to C compiler/linker
-output-obj	link, but output a C object file
-noautolink	do not automatically link C libraries

Errors and Warnings

Warnings default is +a-4-6-7-9-27..29

-w *wlist*	set or unset warnings
-warn-errors *wlist*	set or unset warnings as errors
-warn-help	print description of warnings
-rectypes	allow arbitrarily recursive types

Native-code Specific Arguments

-p	compile or link for profiling with gprof
-inline *size*	set maximal function size for inlining
-unsafe	remove array bound checks

Bytecode Specific Arguments

-custom	link with runtime and C libraries
-make-runtime	generate a pre-customized runtime
-use-runtime *runtime*	use *runtime* instead of ocamlrun

Packing Arguments

-pack -o *file.cmo/.cmx*	pack several units in one unit
-c -for-pack *File*	compile unit to be packed into *File*

Interactive Toplevel

Use ;; to terminate and execute what you typed.
Building your own: ocamlmktop -o unixtop unix.cma

#load "lib.cma";;	load a compiled library/unit
#use "file.ml";;	compile and run a source file
#directory "dir";;	add directory to search path
#trace *function*;;	trace calls to function
#untrace *function*;;	stop tracing calls to function
#quit;;	quit the toplevel

System Variables

OCAMLLIB	Installation directory
OCAMLRUNPARAM	Runtime settings (e.g. b,s=256k,v=0x015)

Flags				
	p	ocamlyacc parser trace	b	print backtrace
	i	major heap increment	s	minor heap size
	O	compaction overhead	o	space overhead
	s	stack size	h	initial heap size
	v	GC verbosity		

Files Extensions

Sources		Objects	
.ml	implementation	.cmo	bytecode object
		.cmx + .o	asm object
.mli	interface	.cmi	interface object
.mly	parser	.cma	bytecode library
.mll	lexer	.cmxa + .a	native library
		.cmxs	native plugin

Generating Documentation

Generate documentation for source files:
ocamldoc *format* -d *directory sources.mli*

where *format* is:		
	-html	Generate HTML
	-latex	Generate LaTeX
	-texi	Generate TeXinfo
	-man	Generate man pages

Parsing   ocamlyacc grammar.mly

will generate grammar.mli and grammar.ml from the grammar specification.

-v   generates grammar.output file with debugging info

%{	Declarations:		
header	%token *token*	%left *symbol*	
%}	%token <*type*> *token*	%right *symbol*	
declarations	%start *symbol*	%nonassoc *symbol*	
%%	%type <*type*> *symbol*		
rules	Rules:		
%%	nonterminal :		
trailer	*symbol* ... *symbol* { *action* }		
		...		
		symbol ... *symbol* { *action* } ;		

Lexing   ocamllex lexer.mll

will generate lexer.ml from the lexer specification.

-v   generates lexer.output file with debugging info

```
{ header }
let ident = regexp ...
rule entrypoint args =
  parse regexp { action }
  | ...
  | regexp { action }
and entrypoint args =
  parse ...
and ...
{ trailer }
```

Lexing.lexeme lexbuf in *action* to get the current token.

Computing Dependencies

ocamldep can be used to automatically compute dependencies. It takes in arguments all the source files (.ml and .mli), and some standard compiler arguments:

-pp *prepro*	call a preprocessor
-I *dir*	search directory for dependencies
-modules	print modules instead of Makefile format
-slash	use \ instead of /

Generic Makefile Rules

```
.SUFFIXES: .mli .mll .mly .ml .cmo .cmi .cmx
.ml.cmo :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.mli.cmi :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.ml.cmi :
        ocamlc -c $(OFLAGS) $(INCLUDES) $<
.ml.cmx :
        ocamlopt -c $(OFLAGS) $(INCLUDES) $<
.mll.ml :
        ocamllex $(OLEXFLAGS) $<
.mly.ml :
        ocamlyacc $(OYACCFLAGS) $<
.mly.mli:
        ocamlyacc $(OYACCFLAGS) $<
```