MBase Reference Manual v1.0.2

Meta Alternative Ltd.

2017

Contents

1 Introduction

MBase is a framework for implementing compilers, code analysis tools and interpreters. It consists of a simple "general purpose" core language and a hierarchy of domain specific languages built on top of the core. This document describes the core language design and a part of the runtime library.

2 Architecture outline

MBase core contains several layers. Each layer forms a complete and usable language, and every next layer is built as an extension to a previous layer functionality. The most basic layer, runtime library and $\mathcal{L}_0$ interpreter, is implemented in C#. This layer is only used for bootstrapping.

2.1 Runtime library

This is a minimalistic library written in C#. It contains a DLL loading interface, a standard bootstrap sequence definition, and a basic runtime functions set, including a minimal comprehensive set of bindings to the `System.Reflection` functionality. Some functions of the runtime could have been omitted and implemented within a higher level code, but they are here for the better $\mathcal{L}_0$ interpretation efficiency. Functions defined via reflection are later redefined as direct calls in a compiled mode (in $\mathcal{L}_1'$).

2.2 $\mathcal{L}_0$ interpreter

It is a closure–based interpreter for the simple $\mathcal{L}_0$ language and an S–expressions parser used by several bottom layers of MBase. $\mathcal{L}_0$ statements are compiled to runnable objects, stacking into one single object for one statement. E.g., `(+ 2 2)` will be represented as two objects returning a constant `2` and an object applying the function `+` to an array of evaluation results of two constant objects.

Expression types are: Try, If, Apply, Sequence, Reference, Constant, Lambda, Closure.

2.3 $\mathcal{L}_1 \rightarrow \mathcal{L}_0$ compiler

This is an $\mathcal{L}_0$ program, compiling $\mathcal{L}_1$ expressions into $\mathcal{L}_0$. It is compiled from $\mathcal{L}_1$ itself, and used for bootstrapping the whole system from scratch. Compiler contains a simple macro expansion preprocessor, which is used later to extend the language. Macros are removed from the bootstrap version of the compiler.

2.4 $\mathcal{L}_1'$ language extensions

After $\mathcal{L}_1$ language is bootstrapped, it is extended (using macro metaprogramming) to the level that provides better usability, including a simple interface to

`.NET` reflection, all standard Lisp–like constructions, some basic pattern matching and lists construction, some basic input/output (via reflected `.NET` libraries).

2.5 $\mathcal{L}_1^{\mathcal{C}}$→CLI compiler

Now, using the functionality of $\mathcal{L}_1'$ language, a "native" CLI compiler is implemented for a superset of $\mathcal{L}_1$: $\mathcal{L}_1^{\mathcal{C}}$. It differs from $\mathcal{L}_1$: recursion and local variables are defined via special constructions, while in $\mathcal{L}_1$ they are macros which expands into more basic forms.

2.6 Alternative target languages

At the level of $\mathcal{L}_1^{\mathcal{C}'}$, one can either use a direct `.NET` code generation infrastructure or target the $\mathcal{L}_1^{\mathcal{C}'}$ or its subsets using macro metaprogramming or an interface to compiler implementation. It is possible to mix both ways, since $\mathcal{L}_1^{\mathcal{C}'}$ have an embedded assebmly and `.NET` class generation macros.

The semantics of some languages requires different target language semantics, often of a much higher level than the raw CLI or the lisp–like $\mathcal{L}_1^{\mathcal{C}'}$. MBase provides some alternative target semantics: it is a Forth–like stack language, Prolog (both interpreted and WAM–compiled), a C#–like `.NET`–specific language, a lazy lambda evaluator (combinator graph reduction based) and a finite state machines builder. Non–`.NET` targets are also available, including an LLVM backend and an extensible C–like language.

3 Metaprogramming support

MBase macro system is somewhat similiar to the Common Lisp one, with some advanced features added.

In Common Lisp–style macro expansion there's only one rule: if a head of a list being processed by an expander is a symbol, and this symbol is a name of a macro, a relevant macro function will be called and its result will be processed by an expander again. Very simple but powerful approach, whereas one important thing is missing: any kind of support for a context. Macros are processed separately and normally can't pass any information to each other.

MBase provides features to handle some sort of a context: lexically scoped local macros and `(inner-expand-first ...)` and `(inner-expand-with ...)` special constructions. There is also an in–list macro syntax.

Local macros are defined as follows:

```
(with-macros
  ((<macro-name> <macro-function>) ...)
  <body>)
```

Macro function is a function of one argument — a list to be macro–expanded. Local macros are valid only in the scope of `<body>`, and macro functions definitions are generated in the compilation time and interpreded by the macro expander (i.e., never compiled and would not appear in a binary module). If a local macro name shadows a global macro name or an outer scope local macro name, the latest (innermost) definition will be used.

This feature can be used to deal with a context. To pass something stored in a local macro to an inner macro, that local macro must be expanded first. This is why we have introduced a special form `inner-expand-first`. For example, an expansion of the following construction:

```
(inner-expand-first mymacro (list 1 2 3))
```

will go into an expansion of `mymacro` with `list` already expanded, i.e., a list `(mymacro (cons 1 (cons 2 (cons 3 nil))))` will be actually expanded.

This little trick allows to feed a macro with a content of a locally scoped macro. One of a practical examples of of this feature application is in the `ast:visit` language defined in an extra libraries section below: local macros are defined for every node and every variant processing code, and macros like `ast:mknode` use that information to substitute a correct format.

`with-macros` itself is a higher level feature not naturally known to a core macro expander. It is a macro built on top of a fundamental macro expansion control form: `(inner-expand-with <hashtable> <code>)`. Hashtable is added to the list of a current context macros name tables and an expansion for an inner code is using this new environment, which is later discarded for an outer scope.

There is no direct access to this environment from macro expanders, so the only way to fetch a data from it is `inner-expand-first`.

4 Library

MBase library is staged as well as MBase itself. The same functionality is provided by different layers, with different levels of abstraction.

4.1 Core library (accessible from $\mathcal{L}_0$)

Core functions are defined in C# library code. Some of them are later overridden by $\mathcal{L}_1^{\mathcal{C}}$ native definitions. The following list is not comprehensive, since we do not want to encourage MBase users to rely on this lowest level.

Function: *car* (l)

Returns a list's head.

Function: *cdr* (l)

Returns a list's tail.

Function: *cons* (l_1 l_2)

Makes a new cons cell of head l_1 and tail l_2.

Function: *null?* (x)

Checks if x is nil.

Function: *list?* (x)

Checks if x is a list.

Function: *string?* (x)
Checks if x is a string.
Function: *symbol?* (x)
Checks if x is a symbol.
Function: *char?* (x)
Checks if x is a char.
Function: *number?* (x)
Checks if x is a number.
Function: *boolean?* (x)
Checks if x is a boolean.
Function: *eqv?* (a b)
Checks if a physically equals to b.
Function: *eq?* (a b)
Checks if a logically equals to b.
Function: > (a b)
Checks if number a is greater than number b.
Function: *mkhash* ()
Makes an empty hashtable.
Function: *hashget* (ht key)
Gets a hashtable value associated with a key.
Function: *hashput* (ht key value)
Puts a value into a hashtable with a given key.
Function: *symbol->string* (value)
Converts a symbol into string
Function: *any->string* (value)
Converts any `.NET` object into string using `ToString` method
Function: *string->symbol* (str)
Makes a symbol out of a given string. Symbol syntax is not enforced.
Function: *string-escape* (str)
Enriches a string with proper escape characters
Function: *getfuncenv* ()
Returns a dictionary with all the current functions namespace definitions
Function: *getmacroenv* ()
Returns a hashtable with all the current macros namespace definitions

4.2 Boot library ($\mathcal{L}_1$ language)

This library is not covered by an automatic documenting system, so here follows a brief description of the most important definitions.
Function: *:Y1* (f)
Y combinator for 1–ary functions.
Usage example:

```
(:Y1 (fun (factorial)
        (fun (n)
           (if (> n 1) (* n (factorial (- n 1))) 1))))
```

Function: *:Y2* (f)
Y–combinator for 2–ary functions.
Function: *:Y0* (f)
Y–combinator for 0–ary functions.
Macro: *:Yn* (n)
Generator for an n–ary Y–combinator.
Macro: *list* args
Makes a list of values
Function: *append* (a b)
Appends a list `b` to the end of the list `a`
Function: *to-string* (value)
Converts a list or atom to string
Function: *read-int-eval* (lst)
Compiles a list into $\mathcal{L}_0$, evaluates it, returns a result
Function: *read-compile-eval-dump* (lst)
Compile a list into $\mathcal{L}_0$, evaluate it, return a string with a compiled code. Supposed to be used for an initial bootstrapping purposes only.

4.3 Essential $\mathcal{L}_1'$ definitions

Here follows some essential definitions that are required by the consequent `.NET` bindings initialisation code, so no logic or structure is present here yet.
Macro: *buildstring* lst
Creates a string builder for a given list of arguments.
Macro: *build-any->string* (arg)
Same as buildstring, but all non–string arguments are wrapped into `any->string`.
Function: *char->string* (ch)
Makes a string of one char.
Macro: *fun* (args . body)
A shorter form for lambda.
Function: @ (f g)
Functional composition: returns a function $\lambda\ x\ .\ f\ (g\ x)$
Function: *iter* (f l)
Imperative iteration, applying `f` to all the `l` elements.
Function: *foldl* (f i l)
Folds `l` with a given `f` and an initial accumulator value `i`.
Function: *foldr* (f i l)
Right-folds `l` with a given `f` and an initial accumulator value `i`.
Function: *filter* (f l)
Filters a list using a given predicate function.
Function: *find* (f l)
Returns the first value conforming to a given predicate or nil.
Function: *lasthead* (l)
Returns the last head of a list or nil.
Function: *flatten* (l)

Returns a flat list of all atoms in `l`.
Function: *first* (i l)
Returns first `i` elements of a given list `l`.
Function: *czip* (a b)
Returns the list of (a_i . b_i) for all elements of `a` and `b`.
Function: *zip* (a b)
Returns the list of (a_i b_i) for all elements of `a` and `b`.
Function: *lasttail* (a)
Returns the last non–`nil` tail of the list `a`.
Function: *iteri* (f l)
Performs an imperative iteration over `l` elements, giving an element number as the first argument to the function `f`.
Function: *mapi* (f l)
Maps `l` elements via f(i,l_i) function, where i is an element number.
Function: *nth* (i l)
Returns an `i`'th element of the list `l`.
Function: *iter-over* (l f)
`iter` with swapped arguments.
Function: *map-over* (l f)
`map` with swapped arguments.
Function: *gensym* ()
Returns a unique symbol every time it is called. Uniqueness is guaranteed within one run only.
Macro: *case* (expr . cases)
Selects an action depending on `expr` symbol value (using `eqv?` to compare).

```
<case>:
   ((<symbol>*) <expression>*)
 | (else <expression>*)
```

Macro: *M@* funs
This is a macro version of the functional composition `@` with an arbitrary number of arguments. E.g., `(M@ f g h)` is equal to
`(fun (x) (f (g (h x))))`.
Macro: *return* (x)
Convinience macro, just expands into `x`.
Macro: *vector* args
Creates a vector of values, an element type is derived from the first value type.

Macro: *ovector* args
Creates an Object vector of given values.
Macro: *writeline* args
Prints a string of arguments into a standard output, using the `to-string` function to print each value.
Function: *writeline* (arg)
Function counterpart for the `writeline` macro. Prints a string to standard output using `to-string` conversion function.

Function: *r_typerx* (tp)
Evaluates the `Type` object for a given symbolic type name.
Macro: *r_typer* (tp)
Expands into the `Type` object evaluation for the given symbolic `.NET` type name.
Macro: *r_mtd* (class method . args)
Expands into a `MethodInfo` for the specified class's method.
Function: *r_mtdf* (class method args)
A function version of the `r_mtd` macro, evaluates a `MethodInfo` object for a given method signature.
Macro: *r_bind* args
Binds a method via reflection, expands into the wrapper function for that method.
Macro: *r_sbind* args
Binds a *static* method via reflection, expands into a wrapper function for that method.
Macro: *r_tbind* args
Expands into a wrapper function for a given `.NET` method, for interpteded mode is the same as `r_bind`.
Macro: *r_tsbind* args
Expands into a wrapper function for a given `.NET` static method, for interpteded mode is the same as `r_sbind`.
Function: *exit* (code)
Exit with a given termination code.
Function: *quit* ()
Exit with a code '0'.
Macro: *new* (classname . args)
Expands into the code creating an instance of a given class, with a given constructor arguments values and types. Here `args` is a list of `(type value)` lists.
Function: *io-open-write* (filename)
Opens a file stream for writing.
Function: *io-open-read* (filename)
Opens a file stream for reading.
Function: *io-open-string* (str)
Opens a string stream for reading.
Definition: *io-read*
Reads an S–expression from the given stream.
Definition: *readline*
Reads a line from the given stream.
Definition: *io-wclose*
Closes an output stream.
Definition: *io-close*
Closes an input stream.
Macro: *include* (fnm)

Expands into the list of values from a given file, enclosed into `(top-begin ...)` statement.
Function: *read-str* (str)
Reads an S–expression from a given string.
From now on, since `include` macro is now defined, the rest of the code appears in a more ordered way.

4.4 `.NET`-specific functionality

In this section some essential `.NET` CLI connectivity features are defined.
Macro: *asetx* (a i b)
A fast array setter, in CLI–mode it is replaced with inline CLI code.
Function: *write* (o)
Calls `System.Console.Write` for a given object.
Function: *->* (vl typ fld)
Gets the field of a given type of `vl`
Function: *:->* (vl fld)
Get the field `fld` of the object `vl`.
Function: *<-* (vl typ fld val)
Set the field `fld` value of the object `vl` into `val`, assuming the given `vl` type `typ`.
Function: *<-:* (vl fld val)
Set the field `fld` value of the object `vl` into `val`. Type of `vl` is evaluated using `GetType`.
Function: *s<-:* (vl fld val)
Sets a property value.
Function: *s->* (typ fld)
Get a static field value.
Function: */->* (vl fld)
Gets a value of the field `fld` of the object `vl`.
Function: *g->* (vl fld)
Gets a property value.
Function: *sg->* (t fld)
Gets a static property value
Macro: *net.types* args
Defines `t_<Typename>` variables for all the given `<Typename>`'s, with no validity checking.
Function: *to_enum_object* (tp obj)
Converts a enum of a given type into an object (`System.Enum.ToObject` method wrapper).
Function: *enum-or* (a b)
Calculates `a` OR `b`, where `a` and `b` are enums.
Function: *a->l* (ar)
Converts an array into a list.
Function: *amap* (f a)
Maps a given `Object` array into an array of the same size via a given function.

Function: *ar->l* (ar)
Converts an instance of `System.Array` into a list.
Function: *getEnum* (tp nm)
Returns a enum of a given type.
Function: *hashmap* (fn ht)
Applies the function `fn`(`key value`) to all the key bindings in a given hashtable, returning a list of application results.
Function: *hashiter* (fn ht)
Applies the function `fn`(`key value`) to all the key bindings in a given hashtable.
Function: *->s* (o)
Calls the `ToString` method of `o`.
Macro: *add-assembly* (nm)
`(add-assembly <filename>)` adds an assembly to the local lookup cache.
Macro: *load-assembly* (nm)
`(load-assembly <name>)` function adds an assembly to the local lookup cache.
Function: *r_lookup* (asm nm)
`(r_lookup <assemblyname> <typename>)` function searches for a type in a given assembly
Function: *dotnet* (t)
Returns a `Type` instance for a given type name, using the local assembly cache for lookup.
Function: *udotnet* (path t)
Returns a `Type` instance for a given type name, using the local assembly cache for lookup.
Macro: *using* (lst . rest)
Adds a list of named assemblies to the current local assembly cache, for an inner context only.
Function: *call-with-input-file* (fn f)
Opens an input stream for the file `fn` and passes it as an argument to `f`. After `f` execution, closes the stream and returns `f` evaluation value.
Function: *read-stream* (fi)
Returns a list of strings from the input stream `fi`. Use it with caution!
Function: *read-file* (fn)
Returns a list of strings from the text file `fn`. Use it with caution!
Function: *call-with-output-file* (fn f)
Opens an output stream for the file `fn` and passes it as an argument to `f`. After `f` execution, closes the stream and returns `f` value.
Function: *fprint* (ostream string)
Writes a string into `ostream`.
Function: *fprintln* (fil str)
Writes a string and an endline into `ostream`.
Function: *process-stream* (fi fn)
Reads a stream `fi` line by line, applying a given function `fn` to each string.
Function: *read-file-list* (fn)

Reads a given file `fn` into a lazy list.
Macro: *cpath* (str)
Builds a proper lookup path for the given relative one. Behaviour is similar to the (include ...) macro.
Function: *sleep* (msec)
Waits for `msec` milliseconds.
Function: *mbaseerror* (ex)
Gets a value bound to `MBaseException` instance `ex`.
Function: *apply* (fn ars)
Applies `fn` to the list of arguments `ars`.
Function: *date-now* ()
Returns the current date string.
Function: *enumOr* (tp lst)
Apply the `bitwise or` to all the listed enum values of the type `tp`.
Function: *string<?* (s1 s2)
`#t` if `s1 < s2`.
Function: *string>?* (s1 s2)
`#t` if `s1 > s2`.
Function: *string=?* (s1 s2)
`#t` if `s1` equals to `s2`.

4.5 $\mathcal{L}_1$ basic macros

Macro: *format* (aarg formt . body)
Binds a pattern to an argument value. No checks are done, the value is expected to conform the format. A pattern language is following:

```
 <pattern>:
    <ident>                   - bind this place to given variable
  | ()                        - ignore the contents
  | (<pattern> . <pattern>)   - patterns for head and tail of the list
```

Macro: *fmt* (formt . body)
Creates a function accepting an argument of a given format, see `format` macro for details.
Macro: *funct* (nm args . body)
Creates a global function accepting an argument of a given format, see `fmt` and `format` for details.
Macro: *fccase* (arg . elts)
Select a format and action using `arg` head value. Format is applied to the `arg` tail.

```
<elt>:
  ((<symbol>*) <format> <expr>*)
```

See `format` macro for details.
Macro: *letf* (fs . body)
Binds values to formats. Usage:

```
 (letf ((<format> <value>)*) <expr>*)
```

Macro: *with-syms* (sl . body)
Binds (gensym)–generated values to the variables listed in sl.
Macro: *for* (aft . body)
Iterates the body expressions with a counter.
Usage:

```
(for (<var> <number-from> <number-to>) <expr>*)
```

Macro: *formap* (aft . body)
Iterates the body expressions with a counter, makes a list of their values.
Usage: beginlstlisting (formap (¡var¿ ¡number-from¿ ¡number-to¿) ¡expr¿*) endlstlisting
Function: *reverse* (lst)
Returns the reversed list.
Function: *fromto* (a b)
Creates a list of numbers from a to b (exclusive)
Macro: *foreach* rest
Iterates over a given list.
Usage:

```
 (foreach (<var> <expr1>) <expr>*)
```

<expr1> value must be a list, the body is evaluated for each list element.
Macro: *foreach-map* rest
Iterates over a given list, making a list of body evaluation values.
Usage:

```
 (foreach-map (<var> <expr1>) <expr>*)
```

<expr1> value must be a list, the body is evaluated for each list element.
Macro: *do* rest
Expands into let expression. Usage:

```
(do <expr> (where (<name> <expr>)*))
```

Function: *debugmacro* (name)
Turns on a debugging output for a given macro.
Function: *split* (fn lst)
Splits lst list with a given predicate function fn. Returns a pair of lists, where the first element is a list of elements for which fn gives #t, and the second one contains all other elements of lst.
Function: *tailsplit* (fn lst)
Same as split, tail recursive version.
Macro: *cut* expression
A simple alternative for currying. Creates a lambda function for a given expression with <>'s substituted as arguments.
E.g., (cut + 2 <>) expands into (fun (x) (+ 2 x)).

Function: *qsort* (cf lst)

Sorts a list using a given comparison function.

Macro: *mqsort* (cf lst)

Sorts a list using a given comparison function.

Function: *interleave* (lst del)

Makes a list of `lst` elements interleaved with `del`'s.

Macro: `<>` args

`(<> arg ... fun)` unrolls into `(fun arg ...)`. It is useful with large function definitions (like AST visitors).

Macro: *S*`<<` args

A short form for `(buildstring ...)`

Macro: *Sm*`<<` args

Same as `(string->symbol (S<< ...))`

Macro: *ifnull* (c v)

If `c` is null, do `v`, otherwise return `c` value.

Macro: *try-some* body

Executes `body` expressions one by one until non–nil value is returned.

Macro: *when* (cnd . body)

Equivalent to (`if cnd` (begin `body`))

Macro: *unless* (cnd . body)

Equivalent to (`if cnd` nil (begin `body`))

Macro: `#`*+1* (expr)

`expr` + 1

Macro: `#`*-1* (expr)

`expr` - 1

4.6 Miscellaneous

Function: *read-some-streams* (fl)

Performs a parallel stream reading, `fl` is a list of input streams.

Function: *strinterleave* (lst str)

Builds a string of lst elements interleaved with str.

Macro: *map-car* (l)

Same as (map car ...)

Function: *map-car* (l)

Same as (map car ...)

Macro: *map-cadr* (l)

Same as (map cadr ...)

Function: *map-cadr* (l)

Same as (map cadr ...)

Macro: *map-cdr* (l)

Same as (map cdr ...)

Function: *map-cdr* (l)

Same as (map cdr ...)

Macro: *[* (idx] arr)

(aget arr idx) wrapper

Macro: *lazy* (ex)
Makes a lazy value of an expression. Evaluation can be forced later with `(lazyref ...)`.
Macro: *lazyref* (v)
Forces an evaluation of a lazy value.
Macro: *with-hash* (names . body)
Defines new hash tables with given names and creates shortcut macros for accessing them. For example:

```
(with-hash (ht) (ht! 'a 1) ht)
```

Macro: *with-ohash* (names . body)
Defines new hash tables with given names and creates shortcut macros for accessing them. For example:

```
(with-ohash (ht) (ht! 'a 1) ht)
```

Macro: *use-hash* (names . body)
Creates shortcut macros for accessing listed hashtables.
Macro: *use-ohash* (names . body)
Creates shortcut macros for accessing listed hashtables.
Function: *ccerror* (arg)
Raises `MBaseException` with a given argument.
Function: *ccwarning* (arg)
Adds a warning to the global list of warnings.
Function: *getwarnings* ()
Returns the current list of warnings.

4.7 Simple parsing combinators

Macro: *p-fail?* (r)
`#t` if parsing result `r` is a failure.
Macro: *p-success?* (r)
`#t` if parsing result `r` is a success.
Macro: *p-result* (r)
Returns a contents of the successful parsing result `r`.
Function: *p-rest* (r)
Returns a remaining stream for the parsing result `r` (either successful or unsuccessful).
Macro: *p-rest* (r)
Returns a remaining stream for the parsing result `r`, macro version.
Macro: *p-mkresult* (d rest)
Makes a successful parsing result with given result value `d` and remaining stream `rest`.
Macro: *p-mkfail* (res rest)
Makes a parsing failure result with a given remaining stream `rest`.
Function: *p<+>* (p1 p2)

Sequence combinator.
Macro: *pm<+>* parsers
Sequence combinator, arbitrary number of parsers.
Function: *p<*`|`*>* (p1 p2)
Variant combinator.
Macro: *pm<*`|`*>* parsers
Variant combinator, arbitrary number of parsers.
Function: *p<!>* (p)
Negation combinator
Function: *p<*>* (p)
Parse none-or-many combinator.
Function: *p<+*>* (p)
Parse-one-or-many combinator.
Function: *p<R>* (p f)
Parsing result processing combinator.
Function: *::* (x)
Convert a list of chars `x` into string, returns a list containing that string.
Function: *wrap* (x)
Creates a list of one element, same as `(list x)`.
Function: *p.eof* (l)
Recognizes an end of input stream.
Function: *p>pred* (pr)
Makes a predicate recogniser, which applies a given predicate `pr` to the first element of an input stream. Fails on EOF.
Function: *p>eq* (v)
Makes an equality recogniser, using `eq?` predicate. Same effect as `(p>pred (cut eq? v <>))`.
Function: *p>chareq* (v)
Makes a character equality recogniser, using `genchar=?` predicate.
Function: *p>touch* (p)
Makes a recogniser which is successful if `p` is successful, discarding `p` application results, and fails otherwise.
Macro: *c*`#` (ch)
Expands into an integer representing the given character code.
Function: *p.any* (l)
An always successful recogniser with no result value.
Definition: *p.*
Recogniser, successful for any non-EOF input stream element. Result contains that one element.
Definition: *p.lcalpha*
Regonises a lower case latin character.
Definition: *p.ucalpha*
Recognses an upper case latin character.
Definition: *p.alpha*
Recognises any latin character.
Definition: *p.digit*

Recognises any decimal digit.
Function: *p>string* (str)
Makes a recogniser for a string.
Macro: *pm>string* (str)
Macro version of `p>string`.
Macro: *pm>chars* (str)
Makes a recogniser which accepts all of the `str`'s chars.
Definition: *p.space*
Recognises any whitespace charater.
Definition: *p.newline*
Recognises a newline character.
Function: *p>token* (tk)
Recognises a given token in an input stream, where tokens are lists:
`(<tokenname> ...)`.
Function: *p<0>* (p)
Makes a parser, which discards a recognition result of `p`.
Macro: *<r>* body
This is an easy to use frontend to recursive descent parsing combinators.
Body format is:

```
<body>:
  <string> - parse a string
  <char>   - parse a char
  ?        - always successful parsing, not moving.
  '<>'     - parse EOF.
  <ident> - parser/recognizer reference
  '<anything> - equality parser on <anything>
  `<expr> - equality parser on <expr> value.
  ! <body>* - not a <body> parser
  / <ident> - token parser
  % <string> - recognises all the characters of a <string>
  _ <body>* - discards <body>* parsing result.
  (?? <body>) - <body> or nothing
  = <expr>* - fall back to literal substitution
  T <body>* - if the <body>* parser is successful,
              return the empty successful result at
              the current input stream position.

  <body> + <body>
  <body> <body>   - sequential parsing
  <body>|<body>   - variant parsing, leftmost option first
  <body> -> <expr> - applies <expr> function to <body> parsing result
  <body> :-> <expr> - applies <expr> function to <body> parsing result,
                      wraps the application result into a list.
  <body> *   - none-or-many occurences of <body>
  <body> +*  - one-or-many occurences of <body>
```

Function: *S->N* (str)
Converts a string into an integer number.
Definition: *p.integer*
Recognises an integer decimal number.
Definition: *p.integer.p*
Recognises an integer decimal number and returns an integer.
Definition: *p.ident*
Recognises an MBase symbol.
Definition: *p.ident.p*
Recognises an MBase symbol, returns a symbol.
Function: *strsplit* (pp str)
Splits a string using a given delimiter regular expression.
Function: *strmatch** (pp str)
Returns a list of all matches of `pp` in `str`.
Function: *strreplace** (pp rstr str)
Returns `str` with all occurences of `pp` replaced with `rstr`.
Function: *strmkreplacer** (pp)
Makes a prepared replacer regular expression out of `pp`.
Macro: *strreplacers** rest
Makes a prepared replacer for a given list of pairs of regular expressions and replacement strings.
Macro: *strreplacers*R* rest
Makes a prepared replacer for a given list of pairs of regular expressions and processing functions.
Function: *strapply** (pp str)
Applies a prepared replacer to the given string.

4.8 Advanced pattern matching

Macro: *p:match* (val . ptns)
The most generic form of pattern matching.
Pattern language is following:

```
<pattern>:
  $<ident>  - binds anything to the given identifier
| <symbol>  - matches the symbol value
| <string>  - matches the string value
| <number>  - matches the integer number value
| =<ident>  - checks if the value equals to a given variable value
| $$L[:<ident>] - matches any list
| $$N[:<ident>] - matches any number
| $$S[:<ident>] - matches any string
| $$M[:<ident>] - matches any symbol
| ($$AS:<ident> . <pattern>) - match a pattern and bind a value to a name
| ($$F[:<ident>] <fun(x)>) - checks if <expr> applied to this node gives #t
| ($$R[:<ident>] . <symbol>*) - matches any of the given symbols
```

```
 | ($$XXX[:<ident>] . <pattern>) - any number of list elements before
          pattern is matched
 | ($$FFF <pattern> <fun(x)>) - applies a pattern and additionally applies a guard function
 | ($$FF[:<ident>] <fun(x)> . <pattern>) - checks if <fun> is not nil, and
              applies pattern to a function value.
              This feature is a tribute to Don Syme's banana brackets.
 | (<pattern> . <pattern>) - matches a cons cell, applies patterns
                             to its contents
```

See `Pattern matching` documentation section for details.

4.9 System.Reflection.Emit frontend

Function: *clr:invoke.method* (tp name flags args)
Invokes a dynamically created method. If `flags` value is null, default (`InvokeMethod|Public|Static`) is used.
Function: *clr:make.assembly* (name)
Makes a new assembly builder with a given name.
Function: *clr:make.strong.assembly* (name version key)
Makes an assembly builder using the given strong name (built of a name, an explicit version and a public key fingerprint).
Function: *clr:emit.class* (mod e)
Emits a class definition `e` into a given dynamic module `mod`. The class format is following:

```
<class>:
  (class <name-string> [(extends <type>)]
    [(implements <type>*)]
    <elt>*)
<elt>:
  (field <name> <type> <attrs>)
  (initfield <name> <attrs> . <bytedata>)
  <class>
  (method (<name> <attrs> <ret-type> (<arg-type>*))
     <instr>*)

<instr>:
   (local <name> <type>)
   (label <name>)
   (lift . <elt>)
   (<InstrName> [<arg>])
<arg>:
   (var <name>)
   (label <name>)
   (method <name>) - for another method of this class
   <anything-else>
```

`<InstrName>`'s are the same as fields of `System.Reflection.Emit.OpCodes` class.
Function: *_ldc_i4* (n)
A shortcut for creating a proper `ldc_i4` instruction.
Function: *_ldarg* (n)
A shortcut for creating a proper `ldarg` instruction
Macro: *alet* (name value . body)
Arc–style `let` construction
Macro: *awith* (namevalues . body)
Arc–style t with construction
Macro: *aif* body
Arc–style t if construction
Macro: *pipeline>* body
Makes a pipeline of one argument functions

4.10 Mutable records

Macro: *rec:def* (nm . fields)
Defines a record type `nm` with a list of `fields`. To create a new record instance, use the constructor function (`nm`.new `<initial-value>`*)), to get field value, use (`nm.field` `<instance>`), Or, alternatively: (`nm`.make `:<fieldname>` `<initial-value>` ...) to set field value, use (`nm.field`! `<instance>` `<value>`).
Macro: *collector* (nms . body)
Initialize a collector context, with given adder and getter names. Usage: `(collector (<adder> <getter>) <body-expression>*)` Inside the body expressions you can use `(<adder> somevalue)` function to collect values in order, and then `(<getter>)` to return the collected list of values.
This macro is particularry useful with AST visitors.
Macro: *with-sequence* (nam . body)
Creates a gensym sequence within the `body` context.

4.11 Basic AST support

Macro: *ast:visit* (name toph . patns)
Makes a visitor function for an AST 'name', starting from the node 'toph'. See `AST` documentation section for details.
Macro: *ast:iter* (name toph . patns)
Makes an iterator function for an AST 'name', starting from the node 'toph'. See `AST` documentation section for details.
Macro: *def:ast* (name incls . defns)
Defines a named AST, inheriting properties from 'incls' and adding new 'defns'. The definition is interpreted and exists in compilation time only.
Macro: *ast:mknode* values
Make a node of a current format. To be used within a visitor or revisitor only.

4.12 An easy lexing wrapper.

Macro: *make-simple-lexer* (name . code)
Makes a simple lexer using the given hints. Available hints are:

```
(ident-or-keyword <regexp> <tokenname>)
           - defines the regexp and token for identifiers
             and keywords.
(ident-exceptions <predicate> <tokenname> ...)
(keywords <token>*)     - list of keywords
(keywords-insensitive <token>*)
           - list of case insensitive keywords
(simple-tokens <string> <tokenname> ...)
           - simple string tokens (other than keywords)
(regexp-tokens <regexp> <tokenname> ...)
           - regular expression tokens (constant literals, etc.)
```

Function: *debug-lexer* (lexer src)
Returns a lexing result or error in a printable format.

4.13 LL(1) parsing

Macro: *bnf-parser* (entrs . bnf)
Defines a parser from BNF-like declaration and a given list of entry points.
Entry points are: `(<entry> <name-to-export>)`

```
<node>:
  (<name> <variant>*)

<variant>:
( (<token>*) <expr>)

<token>:
  <symbol> - recognises a token, binds it to the variable  '$<number>'
  <symbol>:<name> - recognises a token, binds it to a given name
```

Function: *lex-and-parse* (lxer prser src)
For given lexer, parser and string, return the result of parsing. Lexer results are passed to the parser via a lazy list.

4.14 List comprehensions

Macro: <*L*> rest
A list comprehensions macro.
Format:

```
(<L> generator-expression | source-sets*)
```

Usage example:

```
 (<L> (cons x y) | x <- '(a b) | y <- '(a b) & (not (eqv? x y)))
```

4.15 R6RS–style syntax-case

Macro: *syntax-rules* (capt . rules)
An R5RS–alike syntax-rules macro transformer. The only significant difference is the lack of hygiene, which is leveraged by the verb— name—–style templates that explicitly introduces new names.
Macro: *syntax-case* (arg capt . rules)
An R6RS-alike syntax-case implementation, with verb—(syntax ...)— local macros to substitute syntactic templates. There is the same way to deal with new bindings as in verb—syntax-rules—.
Macro: *define-syntax* (nm trans)
An R5RS–compatible define-syntax wrapper to be used with syntax-rules and such.

4.16 Generic register scheduling library

Function: *r3:solve* (texprs)
Solves liveness equations against the given prepared instructions list.
Function: *r3:lgraphs* (exprs)
Builds variables interference graphs for a given list of instructions, for all the variables types found in the code.
Function: *r3:allocateregisters* (registers graphs)
Allocates registers for a given variables dependency graph, using a naive graph colouring heuristical algorithm.

4.17 Compiler

Function: *cc:add-plugin* (part fn)
Add a plugin function to the compilation chain. Possible part names are: core, pre-lift, after-lift, flat, dotnet. Plugin function takes one argument and returns the value of the same format as its argument.
Function: *cc:compile-stage1* (env expr)
Performs the first stage of compilation, taking a Core AST as a source and producing a Flat output. This stage does not depend on any environment and guaranteed to be stable against given source.
Function: *cc:toplevel-devour* (env texpr)
A main interface to the compiler. Takes a source expression and feeds it to the compilation pipeline within a given environment.
Function: *cc:toplevel-devour-transparent* (env texpr)
Same as `cc:toplevel-devour`, but it does not handle any exceptions.
Macro: *cmacro* (name args . body0)
Defines a compilation mode specific macro. Same syntax as for `(macro ...)`.
Function: *read-compile-eval* (lst)
Redefinition of `(read-compile-eval ...)`, now it is a compiler's frontend. It should not normally be used from the user's code, but serves as a default callback for wrappers.

Macro: *n.module* (name . r)
Defines a module with a given t name and type. Default type is dll, other types available are: exe, winexe.
Macro: *n.save* ()
Save the current module to an exe or dll file.
Macro: *n.report* ()
Print a compiler statistics for the current module to the standard output.

4.18 CLI class generation

Macro: *:classwrap* (nm fags . body)
Creates a class with a given name `nm` and attributes `fags`.
`body` format is:

```
(extends <classname>)  - parent class, default is System.Object
(main <methodname> <exetype>) - sets exe file entry point.
(implements <interface>*) - interfaces implemented
(xmethod ...) - CLI method, see Emit definition for reference.
(method (<name> <rettype> <argtype>*)
             <attributes> <lisp-function>) - binds a lisp function
(constr ...) - constructor
(field <name> <type> <attrs>) - class field
```

4.19 Embedded Prolog interpreter

Function: *pprologrules* (str)
Parses prolog rules.
Function: *pprologgoal* (str)
Parses a single prolog term.
Function: *pprologgoals* (str)
Parses a comma separated list of prolog terms.
Function: *prolog-print* (term)
Converts a prolog term into a pretty-printed string.
Function: *prolog-pp-results* (res)
Creates a list of pretty-printed prolog query evaluation results
Definition: *DefaultPrologDB*
Basic prolog definitions: and, or, equals, not, append, ...
Function: *simple-prolog* (xdb goals)
Parses a prolog query and executes it over a given rules database. If xdb is null, uses the default one (`DefaultPrologDB`).
Function: *to-prolog* (lst)
Converts the list-based representation into the correct format. The simplified list-based foramt is following:

```
<term>:
  (quote <symbol>)    -> symbol/0 structure
  <symbol>            -> symbol variable
  (<symbol1> <term>*) -> symbol1(term,...) structure
```

Function: *simple-prolog-l* (xdb goals)
Converts a list of query goals from the simplified list representation, evaluates the query over a given rules database (default if null), returns the results.

4.20 `System.Collections` bindings

Function: *stack:new* ()
New `Stack` instance
Function: *stack:push* (s o)
Push an object on the `Stack`
Function: *stack:peek* (s)
Peek an object on top of `Stack`
Function: *stack:pop* (s)
Pop an object from `Stack`
Function: *stack:count* (s)
Number of elements in `Stack`
Function: *alist:new* ()
New `ArrayList` instance
Function: *alist:new:n* (n)
New `ArrayList` instance, `n` storage space preallocated
Function: *alist:new:l* (l)
New `ArrayList` instance, initialized from the list `l`
Function: *alist:add* (al v)
Add an element to the end of `ArrayList`
Function: *alist:length* (al)
Number of elements in an `ArrayList`
Function: *alist:get* (al n)
Get a numbered element of an `ArrayList`
Function: *alist:set* (al n v)
Set (destructively!) a value of a numbered `ArrayList` element
Function: *alist->a* (al)
Convert an `ArrayList` to `object` array.
Function: *alist->l* (al)
Convert an `ArrayList` to list
Function: *queue:new* ()
New `Queue` instance
Function: *queue:new:n* (n)
New `Queue` instance, `n` storage space preallocated
Function: *queue:new:l* (l)
New `Queue` instance, initialized from the list `l`
Function: *queue:length* (q)
Number of elements in a `Queue`
Function: *queue:add* (q v)
Add an element to the `Queue`

Function: *queue:get* (q)
Get an element of the Queue
Function: *queue:peek* (q)
Peek an object of Queue
Function: *queue->a* (q)
Convert a Queue to object array.
Function: *queue->l* (q)
Convert a Queue to list
Macro: *n.foreach* (h . body)
Usage:

```
(n.foreach (<name> <IEnumerable>) <expr>*)
```

Macro: *n.foreach-map* (h . body)
Usage:

```
(n.foreach-map (<name> <IEnumerable>) <expr>*)
```

4.21 Threads support

This module provides a high level interface to .NET threading.
Function: *threads:thr:mkthread* (fn)
Makes a thread with a given controller function.
Function: *threads:thr:start* (t)
Starts a thread.
Function: *threads:thr:abort* (t)
Aborts a thread's execution
Function: *threads:thr:mkmanual* ()
Makes a manual switch object.
Function: *threads:thr:mkmutex* ()
Makes a mutex object.
Function: *threads:thr:mutex_wait* (mtx)
Waits for a mutex.
Function: *threads:thr:mutex_release* (mtx)
Releses a mutex object.
Function: *threads:thr:manual_wait* (m)
Waits for a manual switch.
Function: *threads:thr:manual_set* (m)
Sets a manual switch.
Function: *threads:thr:manual_reset* (m)
Resets a manual switch.
Function: *threads:thr:mkworker* (threnv bodyfun)
Returns a pair of a thread worker controller function and a message sending function to trigger the execution of the controller.
Function: *threads:thr:mkpool* (endgame)
Makes a thread pool.
Function: *threads:thr:pool-add* (pool)

Adds one new thread to a given pool.
Function: *threads:thr:pool-add-env* (pool env)
Adds one new thread with a given environment to a given pool.
Function: *threads:thr:pool-send* (pool msg)
Sends a message to a given pool. `msg` is a function with one argument.
Function: *threads:thr:pool-kill* (pool)
Terminates a given thread pool after all the outstanding messages are executed.
Function: *threads:thr:mkqueue* (nthr? consumer)
Makes a consumer queue with a given consumer processor function. If `nthr?` is `#t`, makes a dedicated queue controller thread, otherwise uses the current one.
Function: *threads:thr:queue-add* (q v)
Adds a value to the consumer queue.
Function: *threads:thr:queue-kill* (q f)
Kills a given consumer queue `q`, evaluating `f` in the queue's controller context before termination.
Function: *threads:thr:queue-start* (q)
Starts the queue controller (either in the current thread or in a dedicated one).

4.22 XML and SXML support

SXML format is used for internal representation of XML trees. Detailed specification can be found at `http://ssax.sf.net/`.
Function: *xml-read* (nm)
Reads an XML stream from a file into an SXML tree.
Function: *dumpxml* (filnm xml enc)
Dumps a given SXML tree into an XML file.
Macro: *sxml-path* pth
Creates a path extraction function. The path element format is following:

```
<path>:
  (<node>) - all matches of the node on this level
  <node> - first match of the node on this level
  * <path> - first match of the path somewhere deep.
 (*) <path> - all matches of the path in the depth.
 <path> <path> - match the first path, lookup the second
              from that level.
```

Macro: *sxml-paths* pth
Creates an sxml-path function that is applied to a list of SXML trees Useful for stacking sxml-path chunks together. Always returns a list of values.

4.23 NET types handling library

This library is designed mainly for Not.Net target sublanguage.
Function: *il-type-class-int* (ttp)

Classifies the given .NET type (an instance of System.Type) by its storage. Possible values are: I, I1, I2, I4, I8, R4, R8, Ref.
Function: *il-types-assignable* (ttp)
Returns a list of types assignable from a given one: all the interfaces it implements and all its direct ancestors.
Function: *il-types-havemethod* (ltps mtdname rettype signature)
Returns a list of types which have a method of a given name and fits a given signature. If a return type or some of argument types are unknown, they can be null.
Function: *il-type-constructors* (tp signature)
Returns a list of types which have a constructor that fits a given signature. If some of argument types are unknown, they can be null.
Function: *il-types-havefield* (ltps fldname fldtype)
Returns a list of types which have a field of a given name and type.
Function: *il-types-havemethod-refined* (ltps mtdname rettype signature)
Returns a refined result of `il-types-havemethod`, leaving method declaring types only.
Function: *il-types-havefield-refined* (ltps fldname fldtype)
Returns a refined result of `il-types-havefield`, leaving method declaring types only.

4.24 Not.Net target language: low level NET imperative functionality

Macro: *not.net* (args body)
Compiles and substitutes a Not.Net AST body
Macro: *not.net.lift* (xtp blk? args lifts body)
Compiles and substitutes a Not.Net AST body
Macro: *lltnet-macro* (name args . body)
Defines a Not.Net.hlevel macro.
Macro: *not.neth* (args . body)
Compiles and substitutes a Not.Net simple form code.
Macro: *not.nethf* (args . body)
Compiles and substitutes a Not.Net simple form code.
Macro: *not.nethr* (args . body)
Compiles and substitutes a Not.Net simple form code, adds

```
nil
```

at the end..
Macro: *not.function* (name args . body)
Compiles and substitutes a function containing a Not.Net simple form code.
Macro: *not.class* (name . body)
Compiles and substitutes a Not.Net class with methods code in simple form.
Macro: *not-new-array* (type length)
Creates a new array of a given type

4.25 Not.Net language details

Not.Net is a low level .NET language. It can be either used standalone or embedded into MBase code.

4.25.1 Types

Some short type names are defined for convenience: void, int, ptr, short, long, char, byte, float, double, string, object, bool. A special type name 'this' must be used as a reference to the current class (and it works even with an expression–embedded code).
Other types must be named explicitly, as in C#. Since normal MBase `(dotnet ...)` function is used for a type lookup, its current lookup path is taken into account, i.e. one can use `(using (<namespaces>) ...)` construction.

4.25.2 Statements

Not.Net is a statement–based language, so there is a distinction between statements and expressions.
The following statements are defined:
`(begin ...)` executes statements sequentially.
`(quote <symbol>)` defines a label.
`(for ((<symbol:name> <expr:initial> <expr:step>) <expr:condition>) ...)` is a simple looping statement: a variable `<name>` is set as `<initial>`, and until `<condition>` is false, the body statements are evaluated and `<name>` is updated to `<step>` value.
`(while <expr:condition> ...)` loops until `<condition>` is false, condition is checked prior to the body execution.
`(dowhile <expr:condition> ...)` loops until `<condition>` is false, condition is checked after the body execution.
`(foreach (<symbol:name> <expr:initial>) ...)` iterates over any `System.Collection.IEnumerable` collection.
`(goto <symbol:label>)` jumps to a given label.
`(goto-if <expr:condition> <symbol:label>)` jumps to a given label if a condition value is true.
`(goto-if-not <expr:condition> <symbol:label>)` jumps to a given label if a condition value is false.
`(return <expr:value>)` returns a value from the current method.
`(return)` returns from a void method.
`(if <expr:condition> <statement:iftrue> [<statement:iffalse>])` executes `iftrue` statement if condition value is true, and `iffalse` otherwise.
`(try <statement:code> (catch (<type:exception> <symbol:name>) <statement:excode>))` tries to execute `<code>`, and if an `<exception>` is raised, binds it to `<name>` and executes `<excode>`.
`(throw <expr:value>)` throws and exception.
`(<symbol:name> = <expr:value>)` defines a variable with a given initial value. Variable type is same as `<value>` type.

`(<type> <symbol:name> = <expr:value>)` defines an explicitly typed variable with a given initial value.

`(<lvalue> <- <expr:value>)` destructively assigns a value to a given lvalue (e.g., a local variable, a field, an array element).

`(lift-field <field>)` adds a field to the current class, this works for embedded not.net code as well as for complete class definitions.

`(lift-method <method>)` adds a not.net method to the current class, method is defined as in `not.class` language.

4.25.3 Expressions

The following expressions are allowed:

`((<type>) <expr>)` casts an expression value to a given type

`(type <type>)` loads a type token

`(marshal <type> <expr>)` marshals an expression to a given type

`(arr . <*expr>)` builds an array of given elements, array type if defined by the most generic type of all the expressions.

`(arrt <type> . <*expr>)` builds an array of given elements, array type is specified explicitly.

`(mkarr <type> <expr:length>)` build an array of a given type and dynamically evaluated length (`<length>` must be an integer expression).

`(ref <symbol>)`

`(aref <expr:array> <expr:index>)` references to an element of an array, can be either an expression or an lvalue.

`(begin <statement> ... <expr>)` executes a sequence of statements with a final expression.

`(new <type> . <*expr>)` creates a new object or a value of a given type, using an appropriate constructor call.

`(<type> # <symbol:field>)` references to a static field.

`(<expr> # <symbol:field>)` references to a field.

`(<expr> @ <symbol:method> . <*expr:args>)` calls a method.

`(typeof <expr>)` gives a type of a value in runtime.

`(istype <expr> <type>)` checks a type of a value

`(&& <type> @ <symbol:method> . <*type:argtypes>)`

`(&&& <type> <type> <symbol>)`

`(&&& <type> <expr> <symbol>)`

`null`

`true`

`false`

`self`

`<literal>`

`(<binop> <expr> <expr>)`

`(<unop> <expr>)`

4.25.4 Class definition

Class is defined as follows:
`(not.class <name> [(extends <type>)] [(implements <type>)...] ...)`
Class definition body may contain fields, methods and constructors definitions.
Field definition format is: `(field <type> <name> <attribute> ...)`, where attributes can be `(public)`, `(static)`, `(private)`, `(protected)`.
Method definition is: `(method (<attribute> ...) <type> <name> ((<argtype> <argname>) ...) ...)`
Constructor definition is: `(constructor (<attribute> ...) ((<argtype> <argname>) ...) ...)`

Macro: *nrec:def* (nm . fields)
Defines a record type `nm` with a list of `fields`. To create a new record instance, use the constructor function (`nm`.new `<initial-value>`*)), to get field value, use (`nm.field <instance>`), Or, alternatively: (`nm`.make `:<fieldname> <initial-value>` ...) to set field value, use (`nm.field`! `<instance> <value>`).

4.26 Misc stuff

Macro: *mixed-class* (name . body)
(mixed-class name `(extends ...) (implements ...)` ...) defines a class with both Not.Net and MBase method definitions. Similar to t not.class macro, with an addition of t lmethod entry.
Function: *read-lisp-file* (fn)
Read list of s-expressions from a given file
Macro: *not.staticdata* (name type . data)
Initialise a static data array. Only byte and int types are allowed.
Function: *ast-front-lower* (src)
Transform the frontend def:ast format into astlang
Function: *ast-merge-inherited* (ifun src)
Merge the inherited AST bodies, use ifun to fetch the referenced ASTs
Function: *pattern-get-refs* (p)
Get a list of entries mentioned in a pattern
Function: *visitor-build-paths* (ast entry vis)
Build a table of nodes lying on the paths reaching the explicit nodes
Function: *visitor-list-stop-entries* (vis)
An utility function, returns a hash map with stop entries
Function: *visitor-fuse-implicit-implnodes* (ast entry vis)
Extract a list of imlicitly requirede nodes
Function: *visitor-compile-nodedef* (astS astD ndef)
Compile an implicit node matcher
Function: *visitor-fuse-implicit* (astS astD entry vis)
Fuse implicitly required nodes into a visitor
Function: *visitor-fuse* (srcast dstast vis)
Fuse srcast and dstast into a visitor code
Function: *visitor-inject-listnodes* (src)
Inject additional nodes for lists
Definition: **BUILD**
Long version string.
Definition: **BUILD-VERSION**
Current build version.
Definition: **BUILD-OS**
Build OS string.
Macro: *usedll* (nm)
Loads a DLL produced by MBase. Must refer to the same version of runtime.
Macro: *sysdll* (nm)

Loads a system DLL produced by MBase. Must be of the same version (and signed the same way) as the MBase core library.
Macro: *native* imports
Generates a class with native P/Invoke entries. Entry format is:

```
(import dll-name func-name return-type arg-type ...)
```

Optional class name parameter: (classname Namespace.Class)