Magnolisp

Version 6.4

Tero Hasu

April 13, 2016

This is *Magnolisp*, a small, experimental language and implementation. It is experimental in its implementation technique, which is to replace the phase level 0 (runtime) language of Racket with something non-Racket (here: Magnolisp), and translate it into another language (here: C++) for execution.

Magnolisp is inspired by Racket and the likewise experimental programming language Magnolia. Its algebraic language resembles Racket's, and Racket also provides the module and macro systems. The language is restricted in ways similar to Magnolia, with the restrictions designed to support static reasoning about code, and to allow for fairly direct mapping to most mainstream languages.

Magnolisp is intended to explore and demonstrate techniques for source-to-source compilation on top of Racket, not to support writing of useful applications.

1 Magnolisp the Language

`#lang magnolisp` package: `magnolisp`

The Magnolisp language relies on Racket for its module and macro systems. All of Racket may be used for macro programming. The `racket/base` language is provided by default for phase level 1 (compile time).

A small subset of `racket/base` definitions is also available at phase level 0 by default, as these may be used in runtime code, and evaluated as `magnolisp` in the Racket VM. However, only a small subset of Racket can be handled by the Magnolisp compiler, and either the Magnolisp Racket language or the Magnolisp compiler will report errors as appropriate for uncompilable language.

When a `magnolisp` module is evaluated as Racket, any module top-level runtime expressions will also get evaluated; this feature is intended to facilitate testing during development. The Magnolisp compiler, on the other hand, discards top-level expressions, and also any top-level definitions that are not actually part of the program being compiled.

2 Modules and Macros

The Racket `provide` and `require` forms may be used in Magnolisp as normal, also at phase level 0. However, as far as C++ compilation is concerned, these are only used to connect together Magnolisp definitions internally to the compiled program/library. C++ imports and exports are specified separately using the `foreign` and `export` annotations.

For defining macros and macro-expansion time computation, the relevant Racket facilities (e.g., `define-syntax`, `define-syntax-rule`, `begin-for-syntax`, etc.) may be used as normal.

3 Syntactic Forms

```
(require magnolisp/surface)        package: magnolisp
```

The surface syntax of Magnolisp consists of a collection of Magnolisp-specific forms, as well as a selection of supported Racket forms. The `magnolisp/surface` module defines the Magnolisp-specific ones, and the `magnolisp` language exports these at phase level 0.

3.1 Defining Forms

In Magnolisp, it is possible to `define` functions, types, and variables; of these, variable definitions are not allowed at the top level.

As Magnolisp has almost no standard library, it is ultimately necessary to define primitive types and functions (flagged as `foreign`) in order to be able to compile programs that do anything useful. For this reason there is also a `primitives` convenience form for defining multiple `foreign` types and/or functions at once.

```
(define #:type id maybe-annos)
(define id maybe-annos expr)
(define (id arg ...) maybe-annos expr ...)
(define (id arg ...) maybe-annos #:function Racket-expr)
```

The first form defines a type. Presently only foreign types may be defined, and *id* gives the corresponding Magnolisp name. The `foreign` annotation should always be provided.

For example:

```
(define #:type int #:: (foreign))
(define #:type long #:: ([foreign my_cxx_long]))
```

It is acceptable to define a type for a local (lexical) scope, if the type is not referenced elsewhere.

The second form defines a variable with the name *id*, and the (initial) value given by *expr*. A `type` annotation may be included to specify the Magnolisp type of the variable.

For example:

```
> (let ()
    (define x #:: ([type int]) 5)
    (add1 x))
6
```

Any (module) top-level variable definition will not get translated into C++.

The third form defines a function. The (optional) body of a function is a sequence of expressions, which (if present) must produce a single value.

Unlike in Racket, no tail-call elimination may be assumed even when a recursive function application appears in *tail position*.

Functions may be defined locally within another function. A local function may reference free variables from a surrounding lexical scope, as long as they are not used as L-values (i.e., targets of assignment), and as long as the variables are not top-level.

Where the function is declared as foreign, the Magnolisp compiler will ignore any body *expr* ... sequence. When a function without a body is invoked as Racket, the result is #<void>. When a foreign function has a body, it is typically there to "simulate" the behavior of the C++ implementation in the Racket VM. For purposes of simulation it can be useful to make use of the full Racket runtime language; to implement a function body in Racket syntax instead of Magnolisp syntax, enclose the body expression within a begin-racket form.

A function with the export flag in its annotations indicates that the function is part of the public API of a program that includes the containing module. When a function is used merely as a dependency (i.e., its containing module was not specified as being a part of the program), any export flag is ignored.

When a function includes a type annotation, the type expression must specify a function type (see §3.3 "Type Expressions").

For example:

```
(define (identity x)
  x)
(define (five) #:: (export [type (-> int)])
  5)
(define (inc x) #:: (foreign [type (-> int int)])
  (add1 x))
(define (seven) #:: (foreign [type (-> int)])
  1 2 3 4 5 6 7)
(define (nine) #:: (foreign [type (-> int)])
  (define x (seven))
  (define (compute)
    (inc (inc x)))
  (compute))
```

Here, identity must have a single, concrete type, possible to determine from the context of use. It is not a generic function, and hence it may not be used in multiple different type contexts within a single program.

The second `define` form may also be used to define a function (even a top-level one), provided that the "variable initializer" *expr* is a lambda expression. Alternatively, if the annotations indicate that the definition is for a `foreign` function (whose `type` is explicitly given as a function type), then *id* is also taken to bind a function. Any "variables" that are applied in a program are also taken to refer to functions. For `foreign` function definitions of this form, the *expr* would typically be a function (value) for "simulating" the foreign behavior.

For example:

```
> (let ()
    (define two1 #:: ([type (-> int)]) (#%plain-lambda () 2))
    (define two2 (#%plain-lambda () 2))
    (define mul #:: (foreign [type (-> int int int)]) *)
    (define (four) (mul (two1) (two2)))
    (mul (four) (four)))
16
```

The fourth `define` form (with the `#:function` keyword) may likewise be used to define a function, one whose Racket implementation is given as an *Racket-expr*ession (for which Racket syntax is assumed). The arity of the function value that the expression yields must match the number of declared *arg*uments. When this form is used, it is not necessary to give the type of the function, provided that the type can be inferred from context.

```
(declare #:type id maybe-annos)
(declare (id arg ...) maybe-annos)
```

Forms used to declare C++ translation information for types or functions, not to implement them, or to bind the identifier *id*. The binding must already exist.

The first form states that *id* is a type, probably providing annotations for it (declaring types as `foreign` is presently compulsory).

The second form states that *id* is a function, possibly providing annotations for it. The arguments *arg* `...` only serve to specify the arity of the function.

The key difference between `define` and `declare` is that the former binds the identifier, and thus at the same time necessarily specifies any Racket implementation, while the latter does not. That is: Racket's `define` can be used to give a Racket implementation for something; Magnolisp's `define` can be used to give both a Racket implementation (possibly a "non-implementation") as well as C++ translation information for something; whereas `declare` only gives the C++ translation information for something.

```
(typedef id maybe-annos)
```

Defines a type. An alias for `(define #:type id maybe-annos)`.

```
(function (id arg ...) maybe-annos maybe-body)

maybe-body =
           | expr
```

> **NOTE:** This form is deprecated; use `define`, instead.

Defines a function.

```
(var id maybe-annos expr)
```

> **NOTE:** This form is deprecated; use `define`, instead.

Defines a local variable.

```
(primitives definition ...)

definition = [#:type id]
           | [#:function (id arg ...) :: type-expr]
```

Defines the specified types and functions as `foreign` primitives, whose C++ name is assumed to be the same as the Magnolisp name. Any defined functions will have no Racket implementation (or rather, they will have an implementation that does nothing).

3.2 Annotations

```
      maybe-annos =
                  | #:: (anno-expr ...)

        anno-expr = export-anno-expr
                  | foreign-anno-expr
                  | type-anno-expr
                  | ...

 export-anno-expr = export
                  | (export C++-id)

foreign-anno-expr = foreign
                  | (foreign C++-id)

   type-anno-expr = (type type-expr)
```

where:

```
C++-id
```

A valid C++ identifier. When not provided, a default C++ name is automatically derived from the Magnolisp name.

The set of annotations that may be used in Magnolisp is open ended, to allow for additional tools support. Only the most central Magnolisp-compiler-recognized annotations are included in the above grammar.

It is not always necessary to explicitly specify a `type` for a typed Magnolisp definition, as the Magnolisp compiler does whole-program type inference (in Hindley-Milner style). When evaluating as Racket, `type` annotations are not used at all.

For convenience, the `magnolisp` language installs a reader extension that supports annotation related shorthands: `#ap(anno-expr ...) expr` is short for `(annotate (anno-expr ...) expr)`; and `^type-expr` is short for `(type type-expr)`. For example, `#ap(^int) expr` reads as `(annotate ((type int)) expr)`.

```
(foreign C++-id)
foreign
```

An annotation that marks a type or function definition as foreign. That is, it is a primitive implemented in C++. Whether explicitly specified or derived from the Magnolisp name, any `C++-id` must naturally match that of an existing C++ definition.

```
(export C++-id)
export
```

An annotation that marks a function definition as "public". That is, the function is to be part of the Magnolisp API that is produced for the library being implemented. Its declaration will thus appear in any generated header file.

```
(type type-expr)
```

An annotation that specifies the Magnolisp type of a function, variable, or expression.

```
(literal fmt-elem ...)
```

```
fmt-elem = str
         | fmt-expr

fmt-expr = (fmt-insn fmt-obj)

fmt-insn = display
         | write
         | print
         | cxx-str

 fmt-obj = datum
         | type-id
         | str
```

An annotation for type definitions, specifying how to format literal datums of that type. The specification is given as a sequence of elements, which are either string literals or simple formatting expressions. Each `fmt-expr` is translated into a string, and the resulting strings are then concatenated, in order, to get the C++ string encoding for a each literal `datum` of the concerned type. Thus, exact same datums of different types can translate differently.

`str`

A string literal.

`display`

Use Racket's `display` to format the object.

`write`

Use Racket's `write` to format the object.

`print`

Use Racket's `print` to format the object.

`cxx-str`

Format the object as a "regular" C++ string literal, after coercing it into a string. Only non-control ASCII characters are allowed in the string.

```
datum
```

Substituted with the datum of the literal.

```
type-id
```

Substituted with the C++ name of the literal's type.

3.3 Type Expressions

```
type-expr = type-id
          | (-> type-expr ... type-expr)
          | (<> type-expr type-expr ...)
          | (exists type-var-id ... type-expr)
          | (for-all type-var-id ... type-expr)
          | (auto)
```

Type expressions are parsed according to the above grammar, where `type-id` must be an identifier that names a type. The only predefined types in Magnolisp are `Void` and `Bool`, and any others must be defined using `define #:type` or some other type-binding form.

A `type-var-id` is an identifier that gets bound as a type variable for the context of its type expression.

```
(-> type-expr ... type-expr)
```

A function type expression, containing type expressions the function's arguments and its return value, in that order. A Magnolisp function always returns a single value.

```
(<> type-expr type-expr ...)
```

A parametric type expression, containing type expressions for the type's base type and its type parameters, in that order. Type parameters translate as template parameters in C++.

For example:

```
(define #:type stack #:: (foreign))
(define (stack-id x) #:: ([type (-> (<> stack int) (auto))])
  x) ; the C++ type of this x use will be stack<int>
```

```
(exists type-var-id ... type-expr)
(∃ type-var-id ... type-expr)
```

An existential type. Specified by *type-expr*, which should make use of the *type-var-id* type variables. Each type variable is taken to correspond to exactly one concrete type expression, which must be possible to infer.

```
(for-all type-var-id ... type-expr)
(∀ type-var-id ... type-expr)
```

A universal type. The *type-var-id* type variables may take on any set of type assignments, which must be possible to infer based on their use context. (Each application context of a universally typed function may end up with a different assignment to the type variables.)

Only `foreign` functions may be universally typed.

For example:

```
(define #:type Box #:: (foreign))
(define (box v)
  #:: (foreign [type (∀ E (-> E (<> Box E)))]))
(define (unbox box)
  #:: (foreign [type (∀ E (-> (<> Box E) E))]))
```

```
(auto)
```

A type expression that conveys no information about the thing being typed, indicating that it is expected for the type to be inferable based on the contexts of use of the thing. Equivalent to `(exists T T)`.

3.4 Value Expressions

Like Racket (and unlike C++), the Magnolisp language makes no distinction between statements and expressions. Some expressions yield no useful value, however; such expressions conceptually produce a result of type `Void` (such result values do exist at Racket run time, but not at C++ run time). Some expressions yield *no* values (or *multiple* values), but are merely used as a syntactic device, and only allowed to appear in certain contexts.

Magnolisp borrows a number of constructs from Racket (or Scheme). For example, there is a conditional form `(if test-expr then-expr else-expr)`, as well as the derived forms `(when test-expr then-expr ...+)` and `(unless test-expr then-expr ...+)`. The *test-expr* conditional expression must always be of type `Bool`, and whether it holds depends on the "truthiness" of its value, as interpreted in C++ or Racket (as applicable). The branches of an `if` must generally be of the same type, except where the result of the `if` form is discarded. The `when` and `unless` can generally only appear in such result-discarding contexts, as they have an implicit "else" branch of type `Void`.

A `(begin body ...)` form, in Magnolisp, signifies a sequence of expressions, itself constituting an expression. Similarly to Racket, to allow definitions to appear within an expression sequence, `(let () body ...)` should be used instead.

The `(let ([id expr] ...) body ...+)`, `(let* ([id expr] ...) body ...+)`, and `(letrec ([id expr] ...) body ...+)` forms are also available in Magnolisp, but the named variant of `let` is not supported.

The `(set! id expr)` form is likewise available in Magnolisp, supporting assignment to variables. The left-hand side expression `id` must be a reference to a bound variable. (The `id` may naturally instead be a transformer binding to an assignment transformer, in which case the form is macro transformed as normal.)

In Magnolisp, `(void expr ...)` is an expression with no useful result (the result is of the unit type `Void`). Any arguments to `void` are evaluated as usual, but they are not used. The `(values)` form signifies "nothing," and has no result; hence it is an error for `(values)` to appear in a position where the context expects a result. In result expecting contexts, the former may only appear in a 1-value context, and the latter in a 0-value context (there are few in Magnolisp).

The `define` forms may appear in a Racket *internal-definition context* (and not Racket *expression context*). The same is true of `define-values` forms that conform to the restricted syntax supported by the Magnolisp compiler.

```
(cast type-expr expr)
```

Annotates expression `expr` with the type given by `type-expr`. A `cast` is commonly used to specify the type of a literal, which by themselves are generally untyped in Magnolisp. While the literal `"foo"` is treated as a `string?` value by Racket, the Magnolisp compiler will expect to determine the literal expression's Magnolisp type based on annotations. The `cast` form allows one to "cast" an expression to a specific type for the compiler.

For example:

```
> (cast int 5)
5
```

While generally only declarations require annotations, `cast` demonstrates a specific case where it is useful to associate annotations with expressions.

```
(annotate (anno-expr ...) expr)
```

Explicitly annotates the expression `expr` with the specified annotations. May be used to specify annotations for an identifier that is bound using the regular Racket binding forms such as `let`, `let*`, etc.

For example:

```
> (let ([x (annotate ([type int]) 6)])
    x)
6
> (define-values (ten) (annotate ([type int]) 10))

> ten
10
```

```
(if-target name then-expr else-expr)
```

A compile-time conditional expression that depends on the intended execution target. Currently the only meaningful target language *name* is `cxx`, which stands for C++. When code is being compiled for a target matching *name*, only *then-expr* will be included in generated executable code; otherwise it is *else-expr* that will be subject to evaluation in the target environment.

Note that there is no specific support for execution-target-conditional macro expansion in Magnolisp (such conditionality is possible, but Magnolisp itself has no supporting mechanisms for it). Instead, to generate different code for different targets, one may use `if-target` to macro generate code for *all* targets at once (currently only C++ and Racket). The choice of which alternative code fragment to evaluate will be made after Magnolisp programs' macros have been expanded, but still at compile-time, either during source-to-source or bytecode compilation (depending on the execution target).

For example:

```
> (if-target cxx (seven) (five))
5
```

```
(if-cxx then-expr else-expr)
```

A shorthand for `(if-target cxx then-expr else-expr)`.

See also: `begin-racket`, `let-racket`.

3.5 Racket Forms

To include Racket code in a phase level 0 context that is significant to Magnolisp, you may wrap the code in a form that indicates that the code is only intended for parsing as Racket. Code so wrapped must be grammatically correct Racket, but not necessarily Magnolisp. The wrapping forms `begin-racket` and `let-racket` merely switch syntaxes, and have no effect on the namespace used for evaluating the enclosed sub-forms; the surrounding namespace is still in effect. Nesting of the wrapping forms is allowed.

```
(begin-racket Racket-form ...)
```

A Racket form that is equivalent to writing `(begin Racket-form ...)`, and hence not necessarily a Racket expression. Intended particularly for allowing the splicing of Racket definitions into the enclosing context, which is not possible with `let-racket`.

For example:

```
> (require (prefix-in r. (only-in racket/base define)))

> (begin-racket
    (r.define six 6)
    (r.define (one-more x) (let dummy () (+ x 1))))

> (define (eight) #:: (foreign [type (-> int)])
    (one-more (one-more six)))

> (eight)
8
```

```
(let-racket Racket-expr ...)
```

A Racket expression that is equivalent to writing `(let () Racket-expr ...)`. The Magnolisp semantics is to: ignore such forms when at module top-level; and treat them as uncompilable expressions when appearing in an expression position. Uncompilable expressions are acceptable for as long as they are not part of a compiled program, or can be optimized away.

For example:

```
> (define (three) #:: (foreign [type (-> int)])
    (let-racket
      (define-values (x y)
        (let ()
          (values 1 2)))
      (set! x (let dummy () (one-more y)))
      x))

> (three)
3
```

4 Runtime Library

```
(require magnolisp/prelude)    package: magnolisp
```

The Magnolisp language includes a small number of predefined names (that are not syntax). Most notably, the compiler expects expressions of type `Bool` and `Void` in certain contexts, and it also recognizes their identifiers. While the C++ translation semantics of these types are *not* built into the compiler, such semantics *are* predefined by the language. More specifically, the `magnolisp` language uses the `magnolisp/prelude` module as its “runtime library,” one that specifies the C++ runtime names to which these known types correspond. The `magnolisp/prelude` module only contains such compile-time information, and no Racket bindings are exported from it.

```
Bool : any/c
```

A predefined type. The corresponding C++ type is `bool`, and the corresponding C++ constant values are `true` and `false`, respectively.

```
Void : any/c
```

A predefined type. Such values may not actually exist at C++ run time. The corresponding C++ type is `void`.

5 Core Magnolisp

```
(require magnolisp/core)
```
package: magnolisp

There are a small number of Magnolisp-specific names that are treated specially by the Magnolisp compiler. These are bound in the magnolisp/core module, and exported for phase level 0 by the magnolisp language.

5.1 Magnolisp Built-Ins

A boolean expression is simply an expression of type Bool, which is one of the two predefined types in Magnolisp. The other one is Void, which is Magnolisp's unit type (whose values carry no information).

```
Bool : any/c
```

A predefined type. The literals of this type are #t and #f (which are also the only typed literals in the language). All conditional expressions in Magnolisp are of type Bool.

```
Void : any/c
```

A predefined type. There are no literals for Void values, but the Magnolisp core form (void *expr* ...) evaluates such a value, at least conceptually.

5.2 Magnolisp Core Syntax

Magnolisp core syntax is encoded primarily in terms of Racket's core forms. Magnolisp core forms that have no Racket counterpart, however, are encoded in terms of the #%magnolisp variable, which is treated specially by the Magnolisp compiler. The #%magnolisp binding is exported from the magnolisp/core module.

It is possible to define multiple different surface syntaxes for Magnolisp, and these can be defined as libraries similar to the magnolisp/surface syntax definition used by the magnolisp language. All Magnolisp language variants must, however, refer to the same core bindings (i.e., as exported from racket/base and magnolisp/core), as no other bindings are treated specially by the Magnolisp compiler.

```
#%magnolisp : any/c
```

A value binding whose identifier is used to uniquely identify some Magnolisp core syntactic forms. It always appears in the applied-procedure position of a Racket #%plain-app core form. The value of the variable does not matter when compiling as Magnolisp, as it is never used. To prevent evaluation as Racket, all the syntactic constructs exported by `magnolisp` surround `#%magnolisp` applications with a "short-circuiting" Racket expression.

5.3 Fully Expanded Programs

As far as the Magnolisp compiler is concerned, a Magnolisp program is fully expanded if it conforms to the following grammar. Any syntactic ambiguities are resolved in favor of the first matching grammar rule.

A non-terminal nt_{rkt} is as documented for non-terminal *nt* in §1.2.3.1 "Fully Expanded Programs" of the Racket Reference.

A form $form_{ign}$ denotes language that is ignored by the Magnolisp compiler, but which may be useful when evaluating as Racket.

A form $form_{pname\ =\ pval}$ means the form *form* whose syntax object has the property named *pname* set to the value *pval*. A form $form_{pname\ \neq\ pval}$ means the form *form* whose syntax object has the property named *pname* set to some value that is not *pval*. A form $(sub\text{-}form\ ...)_{pname\ =/\neq\ pval}$ may alternatively be written as $(_{pname\ =/\neq\ pval}$ *sub-form* ...).

Anything of the form $id_{id\text{-}expr}$ is actually a non-terminal like *id-expr*, but for the specific identifier *id*.

```
module-begin-form = (#%module-begin mgl-modlv-form ...)

    mgl-modlv-form = (#%provide raw-provide-spec_rkt ...)_ign
                   | (#%require raw-require-spec_rkt ...)_ign
                   | submodule-form_rkt,ign
                   | (begin mgl-modlv-form ...)
                   | (begin-for-syntax module-level-form_rkt ...)_ign
                   | module-level-def
                   | (define-syntaxes (trans-id ...) Racket-expr)_ign
                   | Racket-expr_ign
                   | in-racket-form_ign

  module-level-def = (define-values ()
                       (begin
                         (if Racket-expr_ign
                             (#%plain-app #%magnolisp 'declare
                               id mgl-expr)
                             Racket-expr_ign)
                         (#%plain-app values)))
                   | (define-values (id) mgl-expr)
```

```
             | (define-values (id ...)
                 (#%plain-app values mgl-expr ...))

 Racket-expr = expr_rkt

in-racket-form = Racket-form_'for-target ≠ 'cxx

    mgl-expr = in-racket-form_ign
             | (begin mgl-expr ...+)
             | (begin0 mgl-expr mgl-expr ...)
             | (#%expression mgl-expr)
             | (#%plain-lambda (id ...) mgl-expr ...+)
             | if-target-expr
             | (if Racket-expr_ign
                   (#%plain-app #%magnolisp 'foreign-type)
                   Racket-expr_ign)
             | (if mgl-expr mgl-expr mgl-expr)
             | (#%plain-app void_id-expr mgl-expr ...)
             | (#%plain-app values_id-expr mgl-expr)
             | (#%plain-app values_id-expr)
             | (#%plain-app id-expr mgl-expr ...)
             | ('annotate ≠ #f
                let-values ([() (begin mgl-anno-expr
                                       (#%plain-app values))]
                             ...)
                  mgl-expr)
             | (let-values (bind-in-let ...) mgl-expr ...+)
             | (letrec-values (bind-in-let ...) mgl-expr ...+)
             | (letrec-syntaxes+values
                   ([(trans-id ...) Racket-expr_ign] ...)
                   (bind-in-let ...)
                 mgl-expr ...+)
             | (set! id mgl-expr)
             | (quote datum)
             | (#%top . id)
             | id

 bind-in-let = [(id ...)
                (#%plain-app values_id-expr mgl-expr ...)]
             | [() mgl-expr]
             | [(id) mgl-expr]

if-target-expr = ('if-target = 'cxx
                if Racket-expr_ign mgl-expr Racket-expr_ign)
             | ('if-target ≠ 'cxx
                if Racket-expr_ign Racket-expr_ign mgl-expr)
```

```
     id-expr = id
             | (#%top . id)
             | (#%expression id-expr)

mgl-anno-expr = (if Racket-expr_ign anno-expr Racket-expr_ign)

   anno-expr = (#%plain-app #%magnolisp 'anno
                'type  type-expr)
             | (#%plain-app #%magnolisp 'anno
                (quote id) anno-val-expr)

anno-val-expr = (quote datum)
              | (quote-syntax datum)

   type-expr = id
             | fn-type-expr

 fn-type-expr = (if Racket-expr_ign
                    (#%plain-app #%magnolisp 'fn type-expr ...+)
                    Racket-expr_ign)
```

where:

`id`

An identifier.

`trans-id`

An identifier with a *transformer binding*.

`datum`

A piece of literal data. A `(quote datum)` form is a literal in Magnolisp, and its type must be possible to infer from context. Boolean literals are an exception, as their Magnolisp type is recognized as `Bool`.

`Racket-form`

Any Racket core form.

in-racket-form

Any Racket form that has the syntax property `'for-target` set to some value that is not `'cxx`, meaning that the form is not intended for compilation to C++. These are ignored by the Magnolisp compiler where possible, and it is an error if they persist in contexts where they ultimately cannot be ignored. (The `begin-racket` and `begin-for-racket` forms are implemented through this mechanism.)

if-target-expr

Indicates a choice between two expressions that is conditional on the compilation target language. Where the syntax property `'if-target` is set to the value `'cxx`, the Magnolisp compiler will only compile the first expression. If it is set to some other value (indicating another target language), only the second expression will be compiled. (The `if-target` form is implemented through this mechanism.)

submodule-form

A Racket submodule definition. Submodules are not actually supported by the `magnolisp` language, but the Magnolisp compiler does allow them to appear, and merely ignores them.

anno-expr

An annotation expression, containing an identifier *id* naming the kind of annotation, and an expression specifying the "value" of the annotation. In the generic case, any symbol can be used to name an annotation kind, and any `quote`d or `quote-syntax`ed datum can give the value. Only annotations of kind `'type` are parsed in a specific way.

(*Warning:* For some of the $\mathtt{id}_{\textit{id-expr}}$ non-terminals, the current parser actually assumes a direct *id*.)

6 Evaluation

Programs written in Magnolisp can be evaluated in the usual Racket way, provided that the `#lang` signature specifies the language as `magnolisp`. Any module top-level phase level 0 expressions are evaluated, and the results are printed (as for Racket's `#%module-begin`).

It is also possible to launch a Magnolisp REPL, by issuing the command:

```
racket -I magnolisp
```

7 Magnolisp-Based Languages

```
(require magnolisp/modbeg)
```
package: `magnolisp`

It is possible to implement languages other than Magnolisp that are translatable into C++ using Magnolisp's compiler. To enable this, the language's implementation must conform to the following requirements:

- The macros of the language must target Magnolisp's core syntax.
- It follows that the language must refer to Magnolisp's built-in types, since certain core forms expect said types.
- The language must export Magnolisp's `#%module-begin` macro, or its own variant thereof, one that prepares all the information that the Magnolisp compiler expects.
- Where it is not the `magnolisp/prelude` module that specifies C++ mappings for the language's built-ins and primitives, the name of said module (or modules) must be communicated to the compiler via the language's own `#%module-begin` macro.

For an example of a language targeting Magnolisp core language, see Erda$_{C++}$.

```
(module-begin form ...)
```

An implementation of Magnolisp's `#%module-begin`.

```
(make-module-begin  stx
                   [#:prelude-path prelude-stx]) → syntax?
  stx : syntax?
  prelude-stx : syntax? = #''(magnolisp/prelude)
```

A helper function for implementing Magnolisp-compiler-compatible `#%module-begin` macros. The *`stx`* argument should be syntax for the `(#%module-begin form ...)` macro invocation. The *`prelude-stx`* argument may be used to specify syntax for a list of module paths that should be loaded by the compiler, so that the compiler will know how to translate runtime support names into C++.

8 Compiler API

```
(require magnolisp/compiler-api)        package: magnolisp
```

The Magnolisp implementation includes a compiler targeting C++. The `magnolisp/compiler-api` library provides an API for invoking the compiler.

```
(compile-modules  module-path-v
                  ...
                 [#:relative-to rel-to-path-v])
 → compilation-state?
  module-path-v : module-path?
  rel-to-path-v : (or/c path-string? (-> any) false/c) = #f
(compile-files path-s ...) → compilation-state?
  path-s : path-string?
```

Invoke the compiler front end for analysing a Magnolisp program, whose "entry modules" are specified either as module paths or files. Any specified modules that are not in the `magnolisp` language are effectively ignored, as they do not contain any `exported` Magnolisp definitions. Both functions return an opaque compilation state object, which may be passed to `generate-files` for code generation.

The optional argument `rel-to-path-v` is as for `resolve-module-path`. It is only relevant for relative module paths, and indicates to which path such paths should be considered relative.

Any `path-s` is mapped to a `` `(file ,path-s) `` module path, coercing `path-s` to a string if necessary.

```
(compilation-state? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a compilation state object (as returned by `compile-modules` or `compile-files`), `#f` otherwise.

```
(generate-files  st
                 backends
                [#:outdir outdir
                 #:basename basename
                 #:out out
                 #:banner banner?])  → void?
  st : compilation-state?
  backends : (listof (cons/c symbol? any/c))
  outdir : path-string? = (current-directory)
  basename : string? = "output"
  out : (or/c #f output-port?) = (current-output-port)
  banner? : boolean? = #t
```

Performs code generation for the program whose intermediate representation (IR) is stored in the compilation state *st*. Code generation is only performed with the specified compiler back ends, and for the specified back end specific file types. For instance, to generate both a C++ header and implementation, you may pass *backends* as `'((cxx (parts cc hh)))`. The *backends* argument is an association list with one entry per backend. Passing *out* as `#f` causes code generation into (separate) files; otherwise the specified output port is used. When *out* is a true value, the *banner?* argument indicates whether banners (with filenames) should be printed to precede individual output files. When *out* is `#f`, the *outdir* argument specifies the output directory for generated files. The *basename* string is used as the "stem" for output file names.

9 `mglc`

The compiler can also be invoked via the `mglc` command-line tool, specifying the program to compile. The tool gets installed by invoking `raco setup`. (Alternatively you may just run it as `./mglc` on Unix platforms.)

To compile a program with `mglc`, list the source files of the program as arguments; the program will consist of all the functions in the listed files that are annotated with the `export` flag, as well as any code on which they rely. A number of compiler options affecting compilation behavior may be passed, see `mglc --help` for a list.

An example invocation would be:

```
mglc --stdout --banner --cxx my-program.rkt
```

which instructs the compiler to print out C++ code into standard output, with banners, for the program `"my-program.rkt"`.

10 Example Code

For sample Magnolisp programs, see the "test-*.rkt" files in the "tests" directory of the Magnolisp implementation codebase.

Note that some of the example programs are written in Magnolisp language variants other than magnolisp (which is the only one documented here), but the differences are typically minor and superficial.

11 Source Code

A Git repository of the Magnolisp source code can be found at:

```
https://github.com/bldl/magnolisp
```

12 Installation

Racket version 6.3 or higher is required to run the software. The software has been tested with version 6.4 only. Version 6.2.1 will not work due to Racket 6.3 having a different macro model, and also due to differences in its organization of `unstable` libraries.

The software is installable directly off GitHub with the command:

```
raco pkg install git://github.com/bldl/magnolisp
```

13 License

Except where otherwise noted, the following license applies: