Objective-CL

Objective-C-like syntax for Common Lisp

Bugs

The `type-specifiers` are not defined yet. I need to learn about `ccl` FFI and perhaps add a syntax, or at least improve the reading of `type-specifiers`. Notably, for now there's merely read in the keyword package so we cannot give type specifiers such as: (NSRect) or (NSWindow*).

Motivation

The purpose of this package is to provide a few reader macros implementing a syntax like Objective-C to program with Objective-C FFI such as the `ccl` Objective-C bridge.

The principles of the Objective-C syntax is that it is a small set of extensions over the syntax of the base language (C in the case of Objective-C). Namely:

- message sending expressions are put inside brackets (inspired from Smalltalk block notation), and have basically the Smalltalk message sending syntax.
- class declarations and definitions (interface and implementation) and other Objective-C specific elements use keywords prefixed by the #\@ character.

The later is a little at odd with lisp nature, where every form is an expression, and where parenthesized syntax is prefered. We will therefore provide a more Smalltalk-like way to define classes and methods (while retaining the #\@ character as prefix for some symbols, and as a reader macro to read Objective-C string literals).

Principles

Two reader macros are provided:

- a reader macro bound to #\[is used to parse message sending expressions, just like in Objective-C, but since the underlying language is lisp, sub-expressions starting with parentheses are read just like normal sexps (they may further contain Objective-CL syntax).
- a reader macro bound to #\@ which is used to read:
    - an Objective-C literal strings when followed by a double-quote starting a lisp string.
    - a class or method definition expression, when followed by an opening bracket #\[. The syntax used for these definition expression is similar to the message sending syntax, but it's processed more like a special operator or macro than a real message sending: the sub-expression are evaluated with different rules that depend on the operation. It's called a pseudo-message.
    - a normal lisp symbol otherwise.

¶

These reader macros expand to normal lisp forms, using symbols exported from a portability layer package, nicknamed OCLO, which should be implemented specifically for each Objective-C bridge or FFI. The implementation of this bridge is out of scope of these syntax- providing reader macros.

Message Sending

The syntax is:

```
objcl-message-expr := '[' message-send  ']' .

message-send       := recipient message .
recipient          := sexp | class-name | 'super' | 'self' .
class-name         := objcl-identifier .

message            := simple-selector | compound-selector final-arguments .

simple-selector    := objcl-identifier .
compound-selector  := objcl-identifier ':' sexp compound-selector
                    | objcl-identifier ':' sexp .
final-arguments    := | '(' type-identifier ')' sexp  final-arguments .
type-identifier    := symbol .


-- FIXME type-identifier; perhaps we need:
-- type-identifier   := symbol | symbol sexp .
-- for example: (char *)cString (array (int 10))tenInts ?
-- Check with what is available at the FFI/bridge level.
```

An `objcl-identifier` is a case sensitive identifier that is converted to a lisp symbol according to the rules of Objective-C to Common Lisp identifier translation.

A `sexp` is a normal lisp expression, which might be another message sending bracketed expression (or another Objective-CL form).

There should be no space between the `objcl-identifier` and the colon. After the first `objcl-identifier` in a compound-selector, the remaining `objcl-identifiers` can be absent, in which case the colon must be separated from the previous expression by a space.

When recipient is `super`, an `(oclo:send-super self ...)` form is returned. FIXME document the other forms returned.

Examples:

```
[self update]

[window orderFront:sender]

[array performSelector:(@selector "drawRect:") withObject:rect]

(let ((o [[NSObject alloc] init]))
   [NSArray arrayWithObjects:o (id)o (id)o (id)nil])


'[array performSelector:(@selector "drawRect:") withObject:rect]
→ (OBJC:SEND ARRAY :PERFORM-SELECTOR (@SELECTOR "drawRect:") :WITH-OBJECT RECT)
```

Class definition

Classes are created by sending a `subClass:slots:` pseudo-message to its superclass.

The syntax is :

```
objcl-definition    := '@[' class-definition | instance-method-definition | class-method-definition ']' .

class-definition          := super-class-name 'subClass:' class-name  'slots:' '(' slots ')' .
```

```
class-name         := objcl-identifier .
super-class-name   := objcl-identifier .
slots              := | slot slots .
slot               := lisp-slot | objcl-slot .
lisp-slot          := slot-specifier . -- see clhs defclass.


-- objcl-slot         := ...           -- not defined yet.
-- We'd want some simplified definition, and using Obj-C names.
```

Examples:

```
@[NSObject subClass:SpaceShip
              slots:((position :accessor ship-position :initform (make-position))
                     (speed    :accessor ship-speed    :initform 0.0))]
```

Method definition

Class and instance methods are defined by sending a pseudo-message to the class, either `method:resultType:body:` to create an instance method, or `classMethod:resultType:body:` to create a class method.

The syntax is :

```
objcl-definition    := '@[' class-definition | instance-method-definition | class-method-definition ']' .

instance-method-definition := class-name 'method:' '(' signature ')'
                                    'resultType:' '(' type-identifier ')'
                                          'body:' body .

class-method-definition    := class-name 'classMethod:' '(' signature ')'
                                         'resultType:' '(' type-identifier ')'
                                               'body:' body .

class-name         := objcl-identifier .
signature          := simple-signature | compound-signature final-signature .
simple-signature   := objcl-identifier .
compound-signature := objcl-identifier ':' '(' type-identifier ')' objcl-identifier compound-signature
                    | objcl-identifier ':' '(' type-identifier ')' objcl-identifier .
final-signature    := '&rest' objcl-identifier .
body               := | sexp body .

-- FIXME type-identifier; perhaps we need:
-- type-identifier   := symbol | symbol sexp .
-- for example: (char *)cString (array (int 10))tenInts ?
-- Check with what is available at the FFI/bridge level.
```

There should be no space between the `objcl-identifier` and the colon. After the first `objcl-identifier` in a compound-selector, the remaining `objcl-identifiers` can be absent, in which case the colon must be separated from the previous expression by a space.

Examples:

```
@[SpaceShip classMethod:(shipAtPosition:(Position)aPosition)
             resultType:(id)
                   body:(let ((new-ship [[self alloc] init]))
                           [new-ship setPosition:aPosition]
                           new-ship)]

@[SpaceShip method:(moveToward:(Direction)aDirection atSpeed:(double)velocity)
```

```
        resultType:(id)
              body:(let ((new-pos [[self position] offset:...]))
                     (do-something new-pos)
                     [self setPosition:new-pos])]
```

String literals

The syntax read is:

```
objcl-string-literal := '@"' { character } '"' .
```

A CL string is read (ie. with the same escaping rules as normal CL strings), and an (oclo:@ "string") form is returned.

Examples:

```
@"Untitled"
@"String with \"quotes\" and \\ backslash."
@"String with
new lines"
```