Cecil Highlights

Purely object-oriented language

- objects, methods, fields
- messages
- type-safe, garbage-collected

Closures, a.k.a. first-class lexically-nested functions

Static type system

- fancy polymorphic types
- type declarations & type checking are optional

Modules, encapsulation

- not fully implemented; just stylized comments
  - Diesel has 'em!

Rich standard library

Object declarations

To declare a class, use an `object` declaration

- abstract class: `abstract object`
- concrete (instantiable) class: `template object`

E.g.:

```
abstract object shape;
template object rectangle isa shape;
template object square isa rectangle, rhombus;
template object circle isa shape;
```

An object can have zero, one, or many parents (a.k.a. superclasses)

Note that an object doesn't declare any of its fields or methods; these are separate top-level declarations

Advanced, fun fact: can add new parents (a.k.a. superclasses) to existing classes from the outside

E.g.:

```
abstract object printable;
extend object shape isa printable;
```


Field declarations (omitting type declarations)

To declare that an object contains an instance variable, use a `field` declaration

E.g.:

```
var field center(s@shape) := new_point(0,0);
field width(r@rectangle);
field height(r@rectangle) := r.width;
```

Fields are declared separately from objects

- fields are associated with their containing objects via the `@object` specializer (more later)
- can add new fields to objects externally, e.g. in separate source files!

Must say `var` for assignable field

- immutable by default

Optional default initial value for field

- can be an expression, e.g. computing the field's initial value from the initial values of other fields

Advanced feature: `shared` fields

Method declarations (omitting type declarations)

To declare a top-level procedure or a method or constructor of a class, use a `method` declaration

E.g.:

```
method new_rectangle(w, h) { ... }
method area(r@rectangle) {
  r.width * r.height }
method move_to(r@rectangle, new_center) {
  r.center := new_center; }
method =(r1@rectangle, r2@rectangle) { ... }
```

A method body is a sequence of zero or more statements, then a final expression which is returned as the method's result

Methods are declared separately from objects

- makes top-level procedures and nested methods syntactically the same
  - "receiver" formal is explicit
- can add new methods to objects externally, e.g. in separate source files!

Specializers on formal parameters

Where you want (dynamic) overloading of methods, make formal parameter have *@object* specializer

Regular global (non-overloaded) procedures:

```
method new_rectangle(w, h) { ... }
```

Single dispatching (receiver-oriented methods):

```
method move_to(r@rectangle, new_center) { ... }
method move_to(c@circle, new_center) { ... }
```

Multiple dispatching (multi-methods):

```
method =(s1@shape, s2@shape) { false }
method =(r1@rectangle, r2@rectangle) { ... }
method =(c1@circle, c2@circle) { ... }
```

At run-time, choose single most-specific method with right number of args inherited by dynamic "classes" of arguments

- msg-not-understood if no methods inherited
- msg-ambiguous if specificity not obvious

(Methods with same name but different numbers of arguments are unrelated, i.e., statically overloaded)

Object creation

Create objects by evaluating object constructor *expressions*

- like `object` declaration, but omit object name
- inherit from the template object (a.k.a. concrete class) being "instantiated"
- can provide initial values for fields, or rely on fields' defaults

E.g.:

```
method new_rectangle(w, h) {
  concrete object isa rectangle {
    -- center gets default value
    width := w, height := h } }

method new_square(w) {
  concrete object isa square {
    -- center gets default value
    -- height derived from width by default initializer
    width := w } }
```

A regular method containing an object constructor expression is Cecil's version of a Java constructor

- but cannot inherit constructor code, unfortunately

Kinds of expressions and statements

Constants, e.g.: `3`, `-4`, `5.6`, `"hi there\nbob"`, `'a'`

- all values are regular, first-class objects
  - e.g. 3 is a child of `int`, has methods, receives messages, etc.

Vector constructors, e.g.: `[]`, `[3+x, y*z, f(x)]`

- vectors are regular, first-class objects too

Object constructors, e.g.: **`concrete object isa`** `circle`

`void`: the result of methods that don't return anything

Variable declarations, e.g.:

```
let w := y * z;
let var x := w + f(w);
```

- variables must be initialized at declaration

Assignment stmts, e.g.: `x := y * f(z);`

- cannot assign to formals or non-`var` locals/globals

Messages...

Closures...

Messages

Use standard procedure-call syntax to send a message to zero or more arguments:

```
start_prog()
center(r)
set_center(r, c)
draw(r, window, loc)
```

Infix & prefix syntax:

```
x + - y << z ** q!i
```

- any sequence of punctuation is a legal infix message name
  - methods defined the same for regular, prefix, and infix msgs
- user-defined precedences & associativity

Syntactic sugar supports dot-notation:

```
r.center  ⇔  center(r)
r.set_center(c)  ⇔  set_center(r, c)
r.draw(window, loc)  ⇔  draw(r, window, loc)
```

Syntactic sugar for `set_`*X* messages to look like assignments:

```
r.center := c;  ⇔  set_center(r, c);
```

Accessing fields

Fields are accessed solely by sending messages

- to read a field named `f` of object `o`, send `f` message to `o`
  - invokes the field's "get accessor" implicit method
  - syntactic sugar: `o.f`
- to update a (`var`) field named `f` of object `o` to new value `v`, send `set_f` message to `o` and `v`
  - invokes the field's "set accessor" implicit method
  - syntactic sugar: `o.f := v`

Syntactic sugar makes accessing fields by messages syntactically "natural"

Can access methods as if they were fields, too

Allows fields to be reimplemented as methods & vice versa, and allows fields to be overridden with methods & vice versa, without rewriting callers

No explicit accessor methods or C#-style properties needed

Resends

In overriding method, can invoke overridden method

```
template object visible_rectangle
                                   isa rectangle;
method move_to(r@visible_rectangle,
                 new_center) {
  resend(r, new_center);
  r.redisplay;
}
```

Can use to resolve ambiguities

```
template object square isa rectangle, rhombus;
method area(s@square) {
  resend(s@rectangle) }
```

(Like Java's `super`)

Closures

First-class function objects

Used for:

- standard control structures (`if`, `while`, &, |, etc.)
- iterators (do, `find`, etc.)
- exception handling (`fetch`, `store`, etc.)

Syntax

- **&**(*formals*){ *zero or more stmts*; *result expr* }
  - e.g.: `&(i,j){ x := i + j; x*x }`
- if no formals, can omit **&**()
  - e.g.: `{ print("hi"); }`

Examples of use:

```
if(i > j, { i }, { j })
[3,4,5].do(&(x){ x.print; })
table.fetch(key, { error("key is absent") })
```

Invoke closure by sending `eval` with right number of arguments

```
let cl := &(i){ i.print_line; };
...
eval(cl, 5);
```


Non-local returns

Support exiting a method early with a non-local return from a nested closure

```
{ ...; ^ result }
{ ...; ^ }
```

Example:

```
method find_index(array, value, if_absent) {
  array.do_associations(&(i, v){
    if(v = value, { ^ i });
  });
  eval(if_absent) }

method find_index(array, value) {
  find_index(array, value,
             { error("not found") }) }
```

Static type declarations

Can give type declarations to formals, results, & variables:

```
field length(r@:rectangle):int;
field height(r@:rectangle):int := r.length;
method new_rectangle(w:int, h:int):rectangle {
  ... }
method move_to(r@:rectangle,
               new_center:point):void { ... }
```

`@`: used to simultaneously specialize & give type to formal

`&(int,bool):string` is a closure type

Parameterization

Can parameterize objects, methods, and fields

- method or field implicitly parameterized over all types in its header prefixed by ‘ (backquote)

Can provide (F-bounded) upper bounds for parameter types

```
abstract object collection[T];
abstract object table[Key <= comparable[Key],
                      Value]
            isa collection[Value];
template object array[Value]
            isa table[int,Value];

method fetch(t@:table[‘Key,‘Value],
             k:Key):Value { ... }
method find_key(
     t@:table[‘Key,‘Value <= comparable],
     val:Value,
     if_absent:&():Key):Key {
  t.do_associations(&(k:Key, v:Value){
    if(v = val, { ^ k });
  });
  eval(if_absent) }
```


Standard control structures

```
if(test, { then });
if(test, { then }, { else }) -- returns a value
if_false(...);

test & { other_test }
test | { other_test }
not(test)

loop({ ... ^ ... });

while({ test }, { body });
while_false(...);
until({ body }, { test });
until_false(...);

exit(&(break:&():none){
  ... eval(break); ... });
exit_value(&(break:&(result_type):none){
  ... eval(break, result); ... });
loop_exit(...);
loop_exit_value(...);
loop_exit_continue(&(brk,cnt:&():none){...});
loop_exit_value_continue(&(b:..,c:..){...});
```


Standard collections

`print, print_string, print_line` (everything)

```
abstract collection[T]
  length, is_empty, non_empty
  do, includes, find, pick_any
  copy

abstract unordered_collection[T]
  sets and bags

abstract ordered_collection[T]
  linked lists

abstract table[Key,Value]
  hash tables, association lists

abstract indexed[Value]
                isa table[int,Value],
                    ordered_collection[Value]
  arrays, vectors, strings

abstract sorted_collection[T <= ordered]
  binary trees, skiplists
```

Unordered collections

```
abstract unordered_collection[T]
               isa collection[T]
  add, add_all
  remove, remove_some, remove_any, remove_all

abstract bag[T] isa unordered_collection[T]

template list_bag[T] isa bag[T]
  new_list_bag[T]

abstract set[T] isa unordered_collection[T]
  union, intersection, difference
  is_disjoint, is_subset

template list_set[T] isa set[T]
  new_list_set[T]
template hash_set[T <= hashable] isa set[T]
  new_hash_set[T]
template bit_set[T] isa set[T]
  new_bit_set[T]
```


Ordered collections

```
abstract ordered_collection[T]
             isa collection[T]
  do  (over 1-4 ordered collections in parallel)
  add_first, add_last, remove_first/_last
  ||  (concatenate)
  flatten  (for collections of strings)

abstract list[T] isa ordered_collection[T]
  first, rest
  set_first, set_rest

template simple_list[T] isa list[T]
  cons
concrete nil[T] isa simple_list[T]
```

- cannot add in place to simple lists

```
template m_list[T] isa list[T]
  new_m_list[T]
```


Keyed tables

```
abstract table[Key,Value]
                isa collection[Value]
  do_associations, includes_key, find_key
  fetch, !
  store, set_!, fetch_or_init
  remove_key, remove_some_keys, remove_all

template assoc_table[K,V] isa table[K,V]
  new_assoc_table[K,V]

template hash_table[K <= hashable,V]
                          isa table[K,V]
  new_hash_table[K,V]
```


Indexed collections

```
abstract indexed[T] isa ordered_collection[T],
                        table[int,T];
  first, second, ..., fifth, last
  set_first, ..., set_last
  includes_index, find_index
  pos, contains, swap, sort
```

Fixed length (no `add`, `remove`):

```
template i_vector[T] isa indexed[T]
  new_i_vector[T](len, filler)
  new_i_vector_init[T](len, &(i){ value })
  new_i_vector_init_from[T](c, &(c_i){value})
```

- `[...]` creates an `i_vector`

```
template m_vector[T] isa indexed[T]
  new_m_vector[_init[_from]][T](...)
```

Extensible:

```
template array[T] isa indexed[T]
  new_array[T]()
  new_array[_init[_from]][T](...)
```

`new_`*X*`_init_from` is like ML's `map`

Strings

```
abstract string isa indexed[char]
  to_lower_case, to_upper_case
  copy_from
  has_prefix, has_suffix
  remove_prefix, remove_suffix
  pad, pad_left, pad_right
  parse_as_int, parse_as_float
  print
```

Fixed length:

```
template i_vstring isa string
  new_i_vstring(len, filler)
  new_i_vstring_init(len, &(i){ value })
  new_i_vstring_init_from(c, &(c_i){value})
```

- `"..."` is an `i_vstring`

```
template m_vstring isa string
  new_m_vstring[_init[_from]](...)
```


Other collections

```
template stack[T] isa m_list[T]
  push, pop, top
  new_stack[T]

template queue[T] isa m_list[T]
  enqueue, dequeue
  new_queue[T]

template histogram[T] isa hash_table[T,int]
  new_histogram[T]
  increment

template graph[Node,Edge]

template partial_order[Node]

files, streams, random_streams, time, ...
```