Cecil Highlights

Purely object-oriented language

- objects, methods, fields
- messages
- rich standard object library

Closures

Type-safe
Garbage-collected

Optional static type system
Modules

- neither fully implemented;
  just treat as stylized comments for now

Basic declarations

Objects (a.k.a. classes), with inheritance:

```
object shape;
object rectangle isa shape;
object square isa rectangle, rhombus;
```

Fields (a.k.a. instance variables):

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

- must say `var` for assignable field
- optional default initial value for field

Methods:

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

Declarations are implicitly mutually-recursive

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(l, w) { ... }
```

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
inherited by dynamic classes of arguments

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

Object creation

Create objects by evaluating object constructor expressions

- like object declarations, but omit name
- inherit from “class” object
- can provide initial values for fields,
  or rely on fields’ defaults

E.g.

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

Expressions and statements

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

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

Object constructors: **`object isa`** `rectangle {...}`

Variable declaration:

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

- variables must be initialized at declaration

Assignment stmt: `x := y * f(z)`

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

Empty method returns `void`

Messages...

Closures...

Messages

Standard procedure call syntax:

```
start_prog()
length(r)
set_length(r, new_length)
draw(r, window, loc)
```

Infix & prefix syntax:

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

- user-defined precedences & associativity

Syntactic sugar supports dot-notation:

```
r.length
r.set_length(new_length)
r.draw(window, loc)
```

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

```
r.length := new_length;
```


Accessing fields

Cannot access field directly
Instead, send a message

```
var field length(r@rectangle) := ...;
```
`--` declares `length` & `set_length` accessor methods

```
let r := new_rectangle(...);
...
r.length := r.length + 1;
```

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

Resends

In overriding method, can invoke overridden method

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

Can use to resolve ambiguities

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

Closures

First-class function objects

Used for:

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

Syntax:

```
&(i,j){ x := i + j; x*x }
{ print("hi"); }
```

Examples:

```
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) {
  array.do_associations(&(i, v){
    if(v = value, { ^ i });
  });
  error("not found"); }
```


Static type annotations

Can annotate role of object:

```
abstract object collection;
template object array isa collection;
concrete object empty_array isa array;
```

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

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

- `@:` used to simultaneously specialize & give type to formal
- `object` simultaneously creates type and implementation
- `&(int,bool):string` is a closure type

Parameterization

Can parameterize objects & methods

Can provide upper bounds for parameter types

```
abstract object collection[T];
abstract object table[Key <= comparable,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) }
```

Parameters must be used, even if type decls omitted

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){
  ... eval(break); ... });
exit_value(&(break){
  ... eval(break, result); ... });
loop_exit(...);
loop_exit_value(...);
loop_exit[_value]_continue(
   &(break,continue){...});
```


Standard collections

```
abstract collection[T]
  length, is_empty, non_empty
  do, includes, find, pick_any
  copy
  print, print_string, print_line (everything)

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]
  pair_do, triple_do, reverse_do
  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, ..., fourth, 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})
```

- `[...]` builds 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](...)
```


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[T] 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
```