Cecil

Purely object-oriented language

- all data structures through user-defined objects
  - **classless** object model
  - type safe, garbage collected, implicit pointers, ...
- all operations through user-defined methods
  - methods can dispatch on 0, 1, or several arguments
    ⇒ **multi-methods**
- all instance variables through user-defined fields
- all control structures through user-defined code manipulating first-class, lexically-nested, anonymous closure objects
- invoke methods, fields, constructors, & closures uniformly through messages

Optional polymorphic static type checking, with F-bounded and signature-bounded polymorphism

- no type inference

Module system (only partially implemented)

Rich standard object library

- includes code for what would be built-in in other languages

Procedures and variables

Use `method` to define regular procedures

- last expression returned
- can overload for different numbers of arguments

Use `let` to define local and global variables

- add `var` keyword to allow assignment, otherwise immutable
- always must initialize at declaration

```
let var count := 0;
method foo(a, b, c) {
  count := count + 1;
  let var d := a + b;
  let e := frob(d, c);
  d := d + e;
  d + 5 }
method frob(x, y) { x - frob(y) + 1 }
method frob(x) { - x / 5 }
```


Closures

Code bracketed in braces is a 0-argument closure; evaluation delayed until invoked by `eval`:

```
let closure := { factorial(10) + 5 };
...
eval(closure)    → 3628805
```

To allow arguments to closure, add `&(x,y,z)` prefix; invoke passing extra arguments to `eval`:

```
let closure2 := &(n){ factorial(n) + 5 };
...
eval(closure, 10)  → 3628805
```

Just like `lambda`, `fn`, `\`:

- anonymous
- lexically scoped
- first-class (almost!)

Glitch: returning closures

Regular closures cannot be returned out of their lexically enclosing scope

- a glitch in the Vortex implementation, not the Cecil language
  - can crash Vortex mysteriously
- prevents currying, `compose`, closures in data structures, ...

To be able to do this, use `&&` rather than `&`:

```
method add_x(x) { &&(y){ x + y } }

let add_2 := add_x(2);
let add_5 := add_x(5);

eval(add_2, 4)    → 6
eval(add_5, 4)    → 9
```

Is this a good language design?

Using closures in control structures

All traditional (and many non-traditional) control structures implemented as regular Cecil procedures, with closures passed by callers supporting the necessary lazy evaluation

For simple lazy evaluation:

```
if(test, { then_value }, { else_value })
test1 & { test2 }
while({ test }, { body })
```

For iteration with arguments:

```
for(start, stop, &(i){ body })
do(array, &(elem){ body })
```

For exception handling:

```
fetch(table, key, { if_absent })
```

For continuation-passing style code:

```
compare(i, j, { if_lt }, { if_eq }, { if_gt })
```


An example

```
-- this is a factorial method
method factorial(n) {
   if(n = 0,
      { 1 },
      { n * factorial(n - 1)}) }

-- call factorial here:
factorial(7)
```


Non-local returns

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

- like a `return` statement in C
- like a limited kind of continuation in Scheme

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

Example:

```
method fetch(table, key, if_absent) {
  table.do_associations(&(k, v){
    if(k = key, { ^ v });
  });
  eval(if_absent) }
method fetch(table, key) {
  fetch(table, key, {
    error("key " || print_string(key) ||
          " not found ") }) }

fetch(zips, "Seattle", { 98195 })
```


Objects, methods, and fields

To define a data structure, use `object` declaration

- to instantiate, use `object isa` expression

To add methods to an object, *specialize* the method to the object by adding `@Object` after the first argument (the receiver argument)

To add instance variables, use `field` declarations

- add `var` keyword to allow assignment, otherwise immutable
- fields can be given default initial values at declaration
- fields can be given initial values at instantiation

```
object Point;
var field x(p@Point) := 0;
var field y(p@Point) := 0;
method area(p@Point) { p.x * p.y }
method shift(p@Point, dx, dy) {
  p.x := p.x + dx;
  p.y := p.y + dy; }
method new_point() {
  object isa Point }
method new_point(x0, y0) {
  object isa Point { x := x0, y := y0 } }
```

Overloaded methods and dynamic dispatching

Can overload methods two ways:

- same name but different numbers of arguments
- same name & number of arguments,
  but different specializer objects

```
method area(p@Point) { p.x * p.y }
...
method area(c@ Circle) { pi * square(c.radius) }
```

Specializer-based overloading resolved by using run-time class
of receiver argument
(a.k.a. dynamic dispatching, message sending)

```
method print_area(x) {
  print(area(x));
}
let var p := new_point(3,4);
print_area(p); -- invokes Point's area method
    → 12
p := new_circle(5);
print_area(p); -- invokes Circle's area method
    → 78.539818
```


Fields accessed by messages

Field declarations implicitly produce 1-2 accessor methods:

- get accessor:
  given object, return field contents
- set accessor (for `var` fields):
  given object & new field contents, modify field

Manipulate field contents solely by invoking these methods

```
var field x(p@Point) := 0;
  ⇒
    method x(p@Point) {
      ... fetch p.x's contents, initially 0 ...}
    method set_x(p@Point, new_value) {
      ... update p.x to be new_value ... }
-- increment p.x:
set_x(p, x(p) + 1)
```

For syntactic convenience,
any call can be written using dot notation:

```
p.x               ⇔  x(p)
p.x := p.x + 1    ⇔  set_x(p, x(p) + 1)
p.shift(3,4)      ⇔  shift(p, 3, 4)
```


Inheritance

Make new ADTs from old ones via `isa` inheritance link

- child/parent, a.k.a. subclass/superclass
- inherit all method & field declarations
- can add new fields & methods, specialized on child object
- can override fields & methods

```
object ColoredPoint isa Point;
-- inherit all @Point fields and methods
-- add some new ones:
field color(@ColoredPoint);
method new_colored_point(x0, y0, c0) {
  object isa ColoredPoint {
    x := x0, y := y0, c := c0 } }

let p := new_colored_point(3,4,"Blue");
print(p.color);   → "Blue"
p.shift(2,-2); -- invoke inherited Point method
print(p.x);       → 5
```


Overriding of methods

Parent and child objects can define overloaded methods

If both apply to a call, child's takes precedence
(a.k.a. overrides the parent's)

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}

method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  Display.plot_point(p.x, p.y);
}

let var p := new_point(3,4);
p.draw; -- only Point's draw applies

p := new_colored_point(5,6,"Red");
p.draw; -- both draws apply, but
        -- ColoredPoint's overrides Point's
```

Resends

Often, overriding method includes overridden method as a subpiece

Can invoke overridden method from overriding method using `resend`

Without `resend`:

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}
method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  Display.plot_point(p.x, p.y);
}
```

With `resend`:

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}
method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  resend;
}
```


Overriding of fields

Since fields accessed through accessor methods, can override accessor methods with regular methods, & vice versa

```
object Origin isa Point;
method x(@Origin) { 0 }
method y(@Origin) { 0 }
```

Because fields accessed through messages, like methods, clients can't tell how message implemented

- can differ in different child objects
- can change through program evolution & maintenance

```
let p := ...; -- Point or Origin object
print(p.x);    -- how is x implemented?
```


Factoring

Methods written in parent object can be reused via inheritance in child objects

Behavior of inherited methods can customized for children by invoking operations on receiver formal argument

Example:
shared `area` method depends on `x` & `y` implementation

```
method area(p@Point) { p.x * p.y }
var field x(p@Point);
var field y(p@Point);
method x(@Origin) { 0 }
method y(@Origin) { 0 }
```

Another example:
code in `Point` that calls `draw` will do colored drawing when invoked on `ColoredPoint`s

Customizing like this is key to effectiveness of OO

Classless object model

Most OO languages distinguish objects/instances from classes, distinguish subclassing from instantiation

Cecil doesn't: **classless** (a.k.a. **prototype-based**) object model

Individual objects have their own implementation directly

- methods specialized directly on object, not separate class thing
- objects with methods/fields specializing on them act like classes

Individual object can inherit behavior from other objects

- if no additional customization, then act like an instance
- if new/overriding implementation, then act like a subclass

Class-like objects can be used directly as instances, too

- e.g. the `Origin` object
- useful for constants, enumerated values, `true`, `false`, etc.

Use object creation expression w/ optional initialization instead of special constructor operation

Multiple dispatching

If we can specialize first argument for particular classes, why not any subset of arguments?
⇒ **multi-methods**

```
method =(p1@Point, p2@Point) {
  p1.x = p2.x & { p1.y = p2.y } }

method =(p1@ColoredPoint, p2@ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Resolve overloading by:

- find all applicable methods
- select & invoke single most-specific application method

```
let p1 = new_point(...);
let p2 = new_point(...);
let cp1 = new_colored_point(...);
let cp2 = new_colored_point(...);

print(p1 = p2);   -- only Point×Point applies
print(p1 = cp2);  -- ditto
print(cp1 = p2);  -- ditto
print(cp1 = cp2);-- both apply, CP×CP wins
```


Multi-method overriding

One mm overrides another if:

- for all the other's specializers, the first method's corresponding specializers are equal to or descend from the other's, and
- either:
  - at least one of the first's specializers strictly descends from the other's, or
  - one of the first method's formals is specialized while the other's is not

```
method foo(p1@Point, p2@ColoredPoint)
  overrides
method foo(p1@Point, p2@Point)

method foo(p1@ColoredPoint, p2@ColoredPoint)
  overrides
method foo(p1@ColoredPoint, p2)
```


Ambiguous methods

Two methods may be mutually ambiguous: neither overrides the other

```
method foo(p1@Point, p2)
  ambiguous with
method foo(p1, p2@Point)

method foo(p1@ColoredPoint, p2@Point)
  ambiguous with
method foo(p1@Point, p2@ColoredPoint)
```

It's an error to send a message and find no single most specific application method

- no applicable methods: message not understood
- no single most specific method: message ambiguous

Resolving ambiguities

Can resolve ambiguities by defining an overriding method

- method can do its own thing, or it can use directed resend to invoke one or more of existing methods

```
method foo(p1@ColoredPoint, p2@Point) { ... }
method foo(p1@Point, p2@ColoredPoint) { ... }

method foo(p1@ColoredPoint, p2@ColoredPoint) {
  -- select first foo method above:
  resend(p1, p2@Point) }
```