



Dylan Grand Tour

Chris Page

Head Moose

Styling Moose Industries

headmoose@stylingmoose.com

(also: chris.page@palmOne.com)



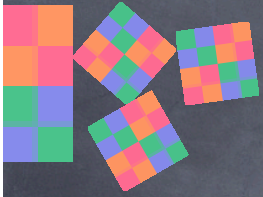
Introduction

- “Dylan” is a portmanteau of **DY**namic **LAN**guage
- Object Oriented: everything is an object, including numbers, classes, and functions
- Safe: arguments and values type-checked, no buffer overruns, no implicit casting, no raw pointers
- Efficient: can compile to code as efficient as C++, or better



2

- Object Oriented: Unlike C++ or Java, all values are objects, including numbers, characters, booleans, types, classes and functions. You can pass these values around or inspect their attributes just like any other object. Every object has a class associated with it at runtime. In Dylan, objects have types, not variables.
- Safe: An error is signaled if some operation doesn't apply to a given object, or if you attempt to access out-of-range elements of a collection like a vector or array. Errors can also be caught at compile time if you give the compiler enough information about your program in the form of type declarations and restrictions on dynamism.
- Efficient: If you provide enough static information in the form of type declarations and restrictions on dynamism, Dylan can be compiled as efficiently as C++, or better. It can also be compiled efficiently using implicit information and static analysis, including type-inference.



History

- Created by Apple Computer in the early 1990's, sold an early version called Apple Dylan TR (Technology Release) in 1995, then ended the project
- CMU created Gwydion Dylan, now open sourced and maintained by Gwydion Dylan Maintainers
- Harlequin, Inc. created Harlequin Dylan for Windows, now called Functional Developer and owned by Functional Objects



3

- Apple Computer: <<http://www.apple.com/>> (Unfortunately, Dylan resources are no longer available at the Apple site.)
- Gwydion Dylan: <<http://www.gwydiondylan.org/>>
- Functional Developer: <<http://www.functionalobjects.com/>>



When I'm learning a new programming language the first thing I like to do is look at some code samples to get a feel for the language, to see how it looks.



Quick Look: hello-world

```
define method hello-world ()  
  format-out( "Hello, World!\n" );  
end method;
```

⇒ Hello, World!



5

This is the seminal “hello, world” program. As you can see, it looks fairly similar to the same program in C/C++, or Pascal. Dylan’s overall syntax should not be surprising to anyone familiar with “Algol-style” languages. I mention this to contrast it with, for example, Lisp and Smalltalk, two popular dynamic languages.

- **define method** introduces a method definition. A method is a kind of function.
- **format-out()** is like C’s printf(), and writes a formatted string to an output stream. In this case, we’re printing a simple string with an end of line (indicated with “\n”), but no other formatting directives or arguments.
- In these slides I’ll use the arrow ⇒ to indicate the output or the results of the code.



Quick Look: factorial

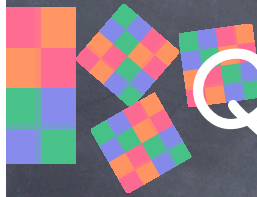
```
define method factorial ( i )  
  if (i = 0)  
    1  
  else  
    i * factorial( i - 1 )  
  end if  
end method;
```



6

This is an implementation of the factorial function. Perhaps the most striking feature here, when compared to C/C++ or Pascal, is that there are no explicit “return” statements. Dylan statements all evaluate to some value, and functions return the value of their last statement or expression.

- Here, if *i* is zero, **factorial()** returns 1, otherwise it returns the result of calling itself recursively.
- Type declarations are optional in Dylan. I’ve omitted them here to emphasize this fact. In fact, notice that there is no declaration of the function result. This means, of course, you could call this function with, say, a string instead of a number and it would signal an error when it attempts to call a math function on it that doesn’t support strings. Type declarations can restrict the values a function can be called with, and help the compiler generate more efficient code.
- Semicolons are optional before “end” (and in this example, “else”). Many Dylan programmers omit the semicolon to indicate when the value of a statement/expression is intended to be returned, and they include the semicolon to indicate that the value is unused.



Quick Look: <point>

```
define class <point> (<object>)  
  slot x = 0;  
  slot y = 0;  
end class;
```

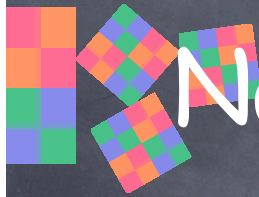
```
let point = make( <point> );  
point.x := 100;  
point.y := 50.2;
```



7

This is a simple class definition. The name of the class is **<point>**. Note that the angle brackets **< >** are part of the name, not special syntax. Dylan allows a wide set of characters in identifiers and there are a number of naming conventions typically used. In this case, classes and types are named with **<** and **>**. I'll cover the naming conventions in more detail later.

- In parenthesis following the class name is the list of superclasses. In this case, **<point>** inherits from **<object>**, the root class.
- Slots are like data members or structure fields in other languages. They define the state or values of instances of a class. The initial value of each slot can be specified in the class definition (unlike C++). Dylan has several ways to initialize slots, including evaluating arbitrary expressions each time an object is created. Here, we just specify the literal value zero for each slot.
- Below the class definition is a code excerpt that shows how you might create an instance and set its slots. **let** introduces a local variable and initializes it (in fact, for safety reasons you can't introduce a local without providing an initial value). **make()** creates an instance of a given class; it's like C++'s **new**, except that it's a function (instead of an operator or a keyword) and can take other arguments to control the creation and initialization of the object.
- Dylan uses **:=** for assignment and **=** for equality comparisons like Pascal and Algol (and unlike C/C++).



Naming Conventions

Names: multiple-words

Types: <object>, <number>, <string>

Globals: *high-score*, *the-port*

Constants: \$pi, \$months-per-year

Predicates: odd?, subclass?, instance?

Mutative: sort!, reverse!



8

Dylan allows a wider set of characters in identifier names than many other languages, and this enables some handy naming conventions. These are only conventions, mind you. The compiler knows nothing about them and does not enforce them.

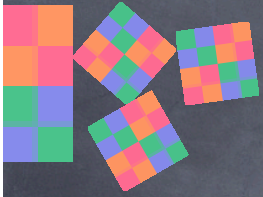
- Hyphen (or minus) is used to separate multiple words in an identifier, where you might use underscore in C/C++ or capital letters in Pascal. Note that this means you have to put spaces around hyphens (and other operators) so they don't get interpreted as part of the operand names. "x - 1" is a subtraction operation, but "x-1" is an identifier.
- Types and classes are surrounded with "angle brackets" (less-than and greater-than).
- Global variables are surrounded with asterisks.
- Global constants begin with a dollar sign.
- Predicates are functions that return a boolean value.
- Mutative functions may destructively modify their input arguments. This naming convention isn't used for all such functions, though. It's generally only used to distinguish mutative functions when there are non-mutative (pure functional) versions available as well. Here, for example, there exist **sort()** and **reverse()** functions that copy the input data into a new collection of objects.



Dylan's big features, including program structure, OOP, dynamism, and language extension (macros).



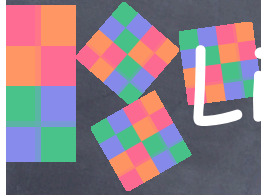
Libraries and modules provide the large-scale organization of Dylan programs.



Libraries

- Dylan programs consist of one or libraries
- Libraries are the unit of compilation and the boundaries of optimization
- Libraries contain one or more modules
- Libraries import modules from other libraries, and export modules for use by other libraries





Library hello-world

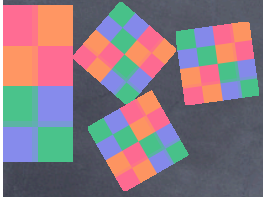
```
define library hello-world
  use dylan;
  use io;

  export hello-world;
end library;
```



12

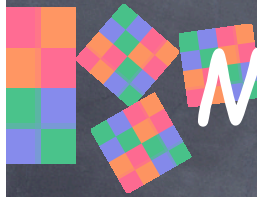
An example of a library definition. The library is named “hello-world”. It imports (uses) all of the modules from the standard “dylan” library and an “io” library. It exports a module named “hello-world”.



Modules

- Modules are namespaces
- Dylan code defines binding names in a module
- Modules import names from other modules, and export names of bindings defined within
- Bindings that aren't exported aren't visible outside the module—modules define “interfaces” and control access to bindings





Module hello-world

```
define module hello-world
  use dylan;
  use format-out;

  export say-hello;
end module;
```



14

An example of a module definition. The module is named “hello-world”. It imports (uses) all the bindings from the standard “dylan” module from the “dylan” library, and from the “format-out” module of the “io” library. It exports a binding named “say-hello”.

- Library and module names are in separate namespaces, so they can have the same name, as in this example, where library “hello-world” has module “hello-world”. They are also in a separate namespace from the names within modules, so the hello-world module could have, say, a function named “hello-world”, although this example doesn’t.
- Libraries and modules are not runtime objects. They exist only at compile-time.



Importing

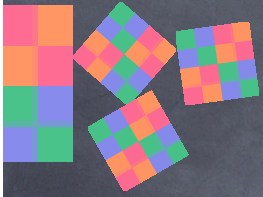
```
use dylan, import: all;  
use streams, import: { <string-stream> };  
use io, exclude: { flush, seek };
```

```
use url-utils, export: all;  
use png-utils, export: { decode-png };
```



15

Library and module definition **use** clauses support several options to import all or only some of the modules/bindings. They can also re-export names that are imported.



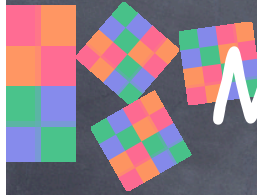
Renaming

```
use dylan, rename: { sort => sort-std };  
use io, prefix: "io-";
```



16

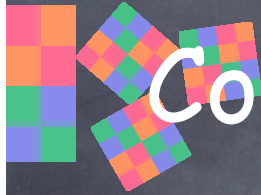
Library and module definition **use** clauses support options to rename modules/bindings upon import. This can be used to resolve naming conflicts or to distinguish names from certain modules. **rename:** renames one or more names. **prefix:** adds a prefix string to every imported name.



Module dylan-user

- All program definitions are in some module
- Libraries & Modules are definitions
- To "bootstrap", compiler defines module "dylan-user", you define your library and its modules in dylan-user, then define your program in one of your library's modules





Complete hello-world

```
module: dylan-user

define library hello-world...
define module hello-world...

module: hello-world

define method say-hello ()
  format-out( "Hello, World!\n" );
end method;
```



Putting it all together to define a Dylan hello-world program.



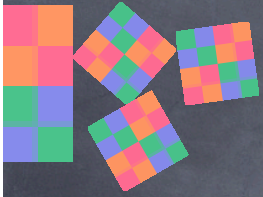
Inside dylan-user

```
define module dylan-user  
  use dylan;  
end module;
```



19

An illustration of what the definition of the “dylan-user” module might look like, were it explicitly defined in your code.



Interfaces

- Many Dylan libraries only contain one module, or perhaps just a few, each of which exports very different things
- More robust libraries may export several modules that represent different "interfaces" on the same functionality, similar to C++'s `public:`, `private:`, and `protected:`, but with more flexibility





Library plug-in

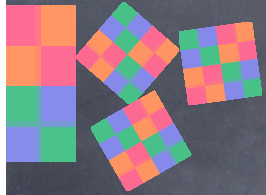
```
define library plug-in
  use dylan;

  export plug-in,
         plug-in-implementor;
end library;
```



21

Let's say you want to define a library for supporting application plug-ins. You would define an interface for plug-ins. However, there may actually be several different interfaces for different needs, e.g.: One for plug-in implementors ("plug-in-implementor"), one for plug-in clients ("plug-in"), and one for the library implementation ("plug-in-implementation"). Perhaps even another for debugging utilities. The implementation module would not be exported and would remain private.



Module plug-in

```
define module plug-in
  use dylan;

  create <plug-in>,
    load-plug-in,
    plug-in-action,
    unload-plug-in;

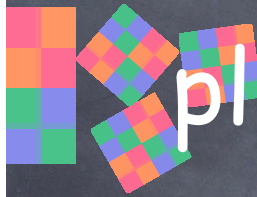
  create plug-in-name;
end module;
```



22

Let's say you want to define a library for supporting application plug-ins. You would define an interface for plug-ins. However, there may actually be several different interfaces for different needs, e.g.: One for plug-in implementors ("plug-in-implementor"), one for plug-in clients ("plug-in"), and one for the library implementation ("plug-in-implementation"). The implementation module would not be exported and would remain private.

- The **create** clause creates and exports a binding without a definition. The implementation module will import these names and provide definitions. We do this to cleanly separate the two modules, so the "public" module doesn't use the implementation module, but the other way around.



plug-in-implementor

```
define module plug-in-implementor
  use dylan;

  create <simple-plug-in>,
    do-load-plug-in,
    do-plug-in-action,
    do-unload-plug-in;
end module;
```



23

The implementation module exports a **<simple-plug-in>** that plug-in writers can subclass. It's different from the public interface class **<plug-in>**, which is probably abstract and has no implementation to inherit. **<simple-plug-in>** might inherit some default behavior from the library.



plug-in-implementation

```
define module plug-in-implementation
  use dylan;

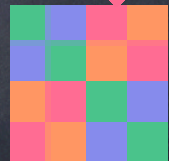
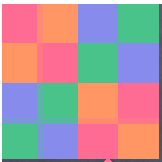
  use plug-in,
        plug-in-implementor;
end module;
```



24

The implementation module uses the public and implementor modules so it has access to all their interfaces and can provide definitions for them.

Classes

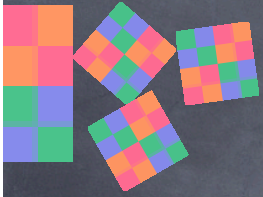




Classes

- A class defines a type in the class hierarchy, and storage for object state, called "slots"
- Every instance object has a class
- Classes do not define "member functions" or a namespace (generic functions and modules fill those roles)

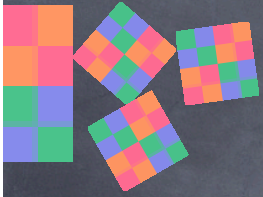




Classes

- Classes are types. There are also types that are not classes.
- There is a root class "<object>". It is the root of the class and type hierarchy.
- Classes are objects; you can pass them around, test properties, etc.
- Dynamic: You can create classes and subclasses at runtime





Class <point>

```
define class <point> (<object>)  
  slot x;  
  slot y;  
end class;  
  
let point = make( <point> );  
unless (slot-initialized?( point, x ))  
  point.x := 0;  
  point.y := 0;  
end;
```



28

This is a simple class definition. We'll grow it over the next few slides to point out slot definition options.

- No initial values are given for the slots, so they're given an implementation-private "uninitialized" value. Any attempt to read that value will signal an error. You can check whether a slot is initialized with **slot-initialized?()**.
- **unless** is like **if**, except it executes the enclosed code if the condition is false (also, it has no "else" or other clauses).



Init Expressions

```
define class <point> (<object>)  
  slot x = 0;  
  slot y = 0;  
end class;
```

```
let point = make( <point> );  
⇒ {<point>: x = 0, y = 0}
```



29

Here, we've added initialization expressions to initialize the slots. They're only zeros here, but arbitrary expressions are supported. There are other options, too, like specifying a function to call to get the initial value.



Init Keywords

```
define class <point> (<object>)  
  slot x = 0, init-keyword: x;;  
  slot y = 0, init-keyword: y;;  
end class;
```

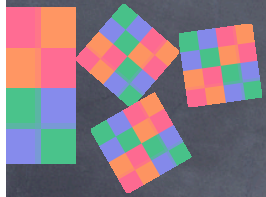
```
let point = make( <point>, y: 42 );  
⇒ {<point>: x = 0, y = 42}
```



30

Slots can also be initialized via keyword arguments to **make()**. Keyword arguments are tagged arguments with a keyword (a <symbol>) and a corresponding value. These are optional, and are applied after handling the initialization expression, so **y** defaults to zero and then is changed to 42 by the **y:** arg to **make()**.

- You can also indicate that a given init keyword argument is required by using **required-init-keyword:** instead, in which case **make()** will signal an error if the arg isn't supplied.
- A **keyword** clause in a class definition can be used to provide further control over initial values, but I've omitted it for brevity.



initialize()

```
define method initialize
  (point :: <point>, #key x-and-y)
  next-method();
  point.x := x-and-y;
  point.y := x-and-y;
end method;

let point = make( <point>, x-and-y: 42 );
⇒ {<point>: x = 42, y = 42}
```



31

Sometimes, you need to do something more complicated to initialize slots, for example if you want to initialize more than one slot from the same init keyword arg, or if you want to enforce some slot value interdependency. After **make()** allocates an instance and initializes it using any available initialization information (from the slot initialization spec, or via init keyword args), it calls the generic function **initialize()**, which you can add a method to for your own classes.

- **next-method()** is an implicit parameter to every method. Calling it calls the next applicable method—the method for a superclass, if any. You should always call **next-method()** early to make sure superclasses perform any initialization they need to before you go mucking with the instance—unless you really know what you’re doing and need to do something before the inherited methods do their thing.



Slot Inheritance

```
define class <point> (<object>)  
  slot x = 0;  
  slot y = 0;  
  slot size = 1;  
end class;  
  
define class <thick-point> (<point>)  
  inherited slot size = 10;  
end class;
```



32

Classes inherit slots from their superclasses. Subclasses can change the initial value for an inherited slot with an “inherited slot specification” that names the slot and provides an init value specification. This can also be used simply to assert that your class inherits such a slot, by declaring an inherited slot without an init specification.

- If a class makes use of multiple inheritance and it inherits a slot through more than one inheritance path, it only inherits the slot once. It’s like C++ “virtual” inheritance.



Slot Types

```
define class <point> (<object>)  
  slot x = 0;  
  slot y :: <integer> = 0;  
end class;
```

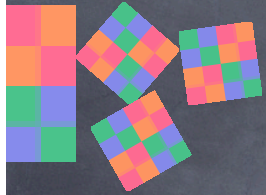
```
let point = make( <point> );  
point.x = "some text";  
⇒ {<point>: x = "some text", y = 0}
```

```
point.y = "some text";  
⇒ {<type-error>}
```



33

As with other bindings in Dylan, you can optionally specify a type for a slot. This constrains the types of values that can be stored in the slot. If you attempt to set it to a value of a different type, an error will be signaled. The default type constraint is **<object>**, the root of the type hierarchy, which allows anything to be stored in the slot.



Slot Allocation

```
instance slot x;  
class slot instance-count = 0;  
each-subclass slot quux;  
virtual slot bar;
```



34

Slots can have one of several “allocation” values that determine where the storage for them is located.

- **instance** allocation is the default. Each instance has storage for the slot.
- **class** allocation slots are allocated one per class. They’re like global variables or C++ **static** class members, except that you access them with the slot accessor functions. They’re only guaranteed to be initialized by the time the first instance is created, by the way.
- **each-subclass** is like **class** allocation, except that, in addition to the class getting storage, each subclass of the class has its own storage for the slot.
- **virtual** allocation doesn’t allocate any storage for a slot. It defines generic functions for accessors, which you must define. This is used for slots whose values are computed, or are stored in other slots or global variables.



Class Adjectives

```
define concrete class...
```

```
define abstract class...
```

```
define sealed class...
```

```
define open class...
```

```
define free class...
```

```
define primary class...
```



35

Class definitions can have “adjectives” that define certain class behaviors. They come in complementary pairs. You can only use one of each pair in a class definition.

- **concrete** classes can have direct instances.
- **abstract** classes cannot have direct instances. If the class can be used as an argument to **make()** to create an indirect instance (an instance of a subclass), then we say the class is “instantiable”, otherwise it is “uninstantiable”.
- **sealed** classes cannot be subclassed by other libraries or at runtime.
- **open** classes can be subclassed by other libraries or at runtime.
- **free** classes can inherit from any other class.
- **primary** classes can only inherit from one other **primary** class. This helps make diamond-shaped inheritance hierarchies more efficient by guaranteeing that slots inherited from multiple superclass paths are all at the same offset for all subclasses. Otherwise, it may incur some runtime overhead to find the location of a slot for particular class instances.
- The defaults are **concrete**, **sealed**, and **free**.



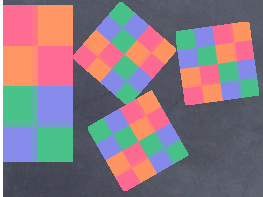
Generic Functions



Generic Functions

- Generic Functions are polymorphic functions
- The basis of polymorphism and implementation inheritance in Dylan
- Contain one or more methods that provide an implementation for specific classes and types
- Dynamic: Methods can be added/removed at runtime
- Are not "owned" by classes





GF Dispatch

- When you call a generic function it dispatches the call to a specific method
- The most-specific method for the argument types will be called
- Multiple Dispatch: The types of all the required arguments are used for method dispatching; there is no distinguished "self" or "this" argument





double()

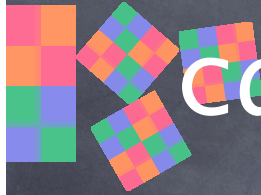
```
define generic double (o);  
  define method double (o :: <object>)  
    pair( o, o )  
  end method;  
  
  define method double (n :: <number>)  
    2 * n  
  end method;  
  
  define method double (s :: <string>)  
    concatenate( s, s )  
  end method;
```



39

This is a simple generic function **double()**. For illustration purposes we've explicitly defined the generic in addition to its methods. Generic functions are also implicitly defined by method definitions, if no explicit definition is provided.

- If you call it with a number or a string, it'll call one of the more specific methods, otherwise, for all other objects it'll call the first method.



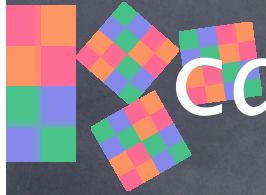
capture?() (1 of 2)

```
define method capture?  
  (a :: <boris>, b :: <moose>)  
  #f  
end method;  
  
define method capture?  
  (a :: <boris>, b :: <squirrel>)  
  #f  
end method;
```



40

capture?() takes two arguments. Generic functions dispatch off the types of all their arguments. There is no distinguished “self” or “this” argument. This function returns true if **a** can capture **b**.



capture?() (2 of 2)

```
define method capture?  
  (a :: <moose>, b :: <boris>)  
  #t  
end method;
```

```
define method capture?  
  (a :: <moose>, b :: <natasha>)  
  #t  
end method;
```



The background is a dark grey chalkboard with faint, illegible text. Scattered around the edges are several colorful squares, each divided into a 3x3 grid of smaller squares in shades of red, green, blue, and orange. The text "Object-Oriented Programming" is written in the center in a white, rounded, sans-serif font.

Object-Oriented Programming



OOP in Dylan

Modules:

- Interfaces
- Access Control

Generic Functions:

- Polymorphism
- Behaviors
- Algorithms

Classes:

- Inheritance
- Types
- Attributes

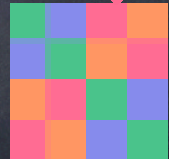
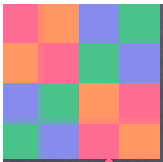


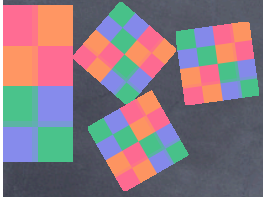
43

- Modules define interfaces and access control.
- Classes define types, attributes, and inheritance.
- Generic Functions define polymorphic behaviors and algorithms.

In contrast, C++ class definitions provide a type and data members, a namespace, three kinds access control (public, private, protected), and member functions (which can be either virtual or non-virtual). Dylan takes a simpler approach with a few orthogonal pieces that can be used in combination with greater flexibility and generality.

Types

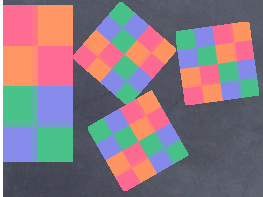




Types

- A type is a set of one or more values
- Classes are types
- There is a root class "<object>". It is the root of the class and type hierarchy.
- Types are objects; you can pass them around, test properties, etc.
- Dynamic: You can create types at runtime





Singletons

- A singleton is a type with only one value; it is used to indicate a single object
- Singleton types can be created with the `singleton()` function
- It is not the singleton pattern, where instantiating a singleton class always returns the one object





singleton()

```
define constant <just-42> = singleton(42);  
instance?( 42, <just-42> );  
⇒ #t  
instance?( 0, <just-42> );  
⇒ #f
```



47

singleton() is a function that creates a type. It takes a single argument and returns a type whose only value is that one object.



==

```
define method fact (n == 0)
  1
end method;
```

```
define method fact (n :: <integer>)
  n * fact( n - 1 )
end method;
```

```
fact(0); // calls the first method
⇒ 1
```

```
fact(3); // calls the second method
⇒ 6
```



48

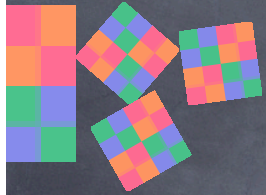
There's a convenient syntax for specializing a method on a singleton "==" . **type == value** is equivalent to writing **type :: singleton(value)**.



Union Types

- A union type is a type whose values include all the values of two or more other types
- Union types can be created with the `type-union()` function
- Particularly useful for allowing values of disparate types without subclassing, e.g., "an rgb color, or a color table index, or a crayon color name string"





type-union()

```
define constant <speed> =  
  type-union(<integer>, <symbol>);  
  
define variable *speed* :: <speed> = 0;  
*speed* := 93;  
*speed* := #"fast";  
*speed* := #"medium";  
*speed* := #t;  
⇒ {<type-error>}
```



50

type-union() creates a type whose values are the values of every argument type. In this example, we define a type that's the union of **<integer>** and **<symbol>**, allowing us to specify the "speed" of things using either a number or a descriptive symbol. Setting to a value that is not one of those types signals an error.



false-or()

```
define method false-or (type :: <type>)  
  type-union( singleton( #f ), type )  
end method;
```

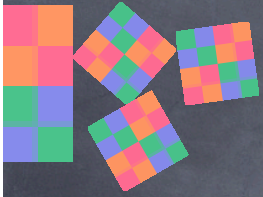
```
let x :: false-or(<string>) = #f;
```

```
if (x)  
  x  
else  
  x := "some text"  
end if;
```



51

false-or() is a common function used to define a type that includes **#f** and the values of some other type or types. It makes use of both **singleton()** and **type-union()**. It's convenient to use this to indicate a binding that can hold, say, a string or "no string" (represented by false).



Limited Types

- A limited type is a type whose values are restricted to some subset of another type
- Limited types can be created with the `limited()` function
- Limited integers can be used to represent subranges of integers
- Limited collections can be restricted in the types of objects they can contain, and they can be length limited





Limited(<integer>)

```
define constant <movie-rating> =  
  limited( <integer>, from: 1, to: 10 );  
  
let rating :: <movie-rating> = 10;  
  
if (has-car-chases?( movie ))  
  rating := rating + 1;  
end if;  
⇒ {<type-error>}
```



53

limited(<integer>) creates a type whose values are a limited subset of integers. Besides increasing type-safety and preventing out-of-range values, this allows Dylan to pick the right implementation of integer to match the required range of values, rather than requiring you to specify a particular integer size.



limited(<collection>)

```
define constant <integer-vec> =  
  limited( <vector>, of: <integer> );  
  
let ratings = make( <integer-vec> );  
  
ratings[0] := 10;  
ratings[1] := 5;  
ratings[2] := "A laugh riot!";  
⇒ {<type-error>}
```

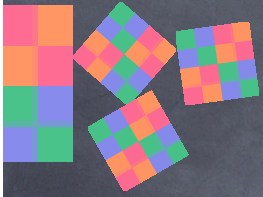


54

limited(<collection>) creates a collection type whose elements are restricted to a given type, and/or whose length is limited (using the optional **size:** argument). In this example, we define a limited **<vector>** whose elements are integers. Besides being more type-safe, this potentially increases performance and reduces memory overhead for storing the integers in the vector, since it needn't check types when reading the vector, and it needn't store type information for every element.

Sealing

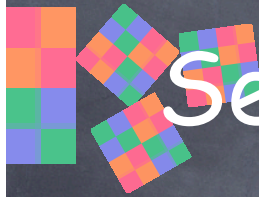




Sealing

- Places limits on dynamism, both at runtime and compile time
- Reduces or eliminates runtime dispatch, type-checking, and other overhead
- You can seal domains, generic functions, methods, classes, and slots
- Libraries are the boundaries of sealing





Sealing Declarations

```
define sealed domain
  capture?(<agent>, <hero>);
define sealed generic foo (x, y);
define sealed method foo
  (<moose>, <squirrel>) ...
define sealed class <boris> (<agent>) ...
define class <natasha> (<agent>)
  sealed slot s;
end class;
```



57

This shows how to declare sealed domains, generics, methods, classes, and slots. Sealing disallows specific kinds of runtime dynamism and specific kinds of compile-time definitions across libraries.

- Sealed domains disallow adding more-specific methods or adding subclasses of the argument types outside the defining library or at runtime.
- Sealed generic functions define a generic function and a sealed domain for the entire generic function.
- Sealed methods define a method and a sealed domain for their argument types.
- Sealed classes can't have subclasses added outside the defining library or at runtime.
- Sealed slots define sealed accessor methods (which define sealed domains).



Conditions & Exceptions

Dylan has a very powerful system for handling errors (exceptions) and other conditions.

[SLIDES NOT YET WRITTEN]



Dylan supports language extension via hygienic macros that operate on parser tokens rather than raw text like preprocessor macros.

[SLIDES NOT YET WRITTEN]



The details of Dylan syntax and basic semantics.



Dylan Basics

The fundamentals of Dylan syntax and semantics.



Naming Conventions

Names: multiple-words

Types: <object>, <number>, <string>

Globals: *high-score*, *the-port*

Constants: \$pi, \$months-per-year

Predicates: odd?, subclass?, instance?

Mutative: sort!, reverse!



62

Dylan allows a wider set of characters in identifier names than many other languages, and this enables some handy naming conventions. These are only conventions, mind you. The compiler knows nothing about them and does not enforce them.

- Hyphen (or minus) is used to separate multiple words in an identifier, where you might use underscore in C/C++ or capital letters in Pascal. Note that this means you have to put spaces around hyphens (and other operators) so they don't get interpreted as part of the operand names. "x - 1" is a subtraction operation, but "x-1" is an identifier.
- Types and classes are surrounded with "angle brackets" (less-than and greater-than).
- Global variables are surrounded with asterisks.
- Global constants begin with a dollar sign.
- Predicates are functions that return a boolean value.
- Mutative functions may destructively modify their input arguments. This naming convention isn't used for all such functions, though. It's generally only used to distinguish mutative functions when there are non-mutative (pure functional) versions available as well. Here, for example, there exist **sort()** and **reverse()** functions that copy the input data into a new collection of objects.



Literal Constants

Number: 123, -1.5e3, #x1fde, #b110, #o777

Character: 'A', '\n', '\\', '\'

String: "text", "two\nlines", "\<44>"

Symbol: foo:, bar:, #"red", #"black"

Boolean: #t, #f

Pair: #(1 . 2)

List: #(1, 2, 3)

Vector: #[1, 2, 3]



63

- Numbers can be positive or negative, integers or floating point. Floating point numbers can use exponential notation. Integers can be given in hexadecimal (#x), binary (#b), or octal (#o).
- Like C, characters and strings can contain special characters indicated with backslash, like '\n' (newline). They can also contain hexadecimal Unicode character values, e.g., "\<44>\<79>\<6c>\<61>\<6e>" = "Dylan". Backslash can be used to escape backslash and single and double quotes.
- Symbols are unique strings. They are case-insensitive. There are two literal syntaxes for them, **keyword:** and **#"unique string"**. **foo:** and **#"foo"** specify the same symbol value. **keyword:** syntax is useful for places where you have a **keyword:** followed by a value associated with the keyword. **#"unique string"** syntax allows more characters, such as spaces and colons.
- **#t** is true, and **#f** is false. We often pronounce them as just "tee" and "eff".

Dylan has a rich set of collection classes—objects that contain other objects. These literals can only contain other literal constants, not arbitrary expressions or function calls.

- A pair is a collection with two values, **head** and **tail**. Some people might call it a "two-tuple". It's like a Lisp cons cell, and can be used on its own or to construct lists or other linked data structures.
- A list is a linked-list.
- A vector is a one-dimensional array.



Operators

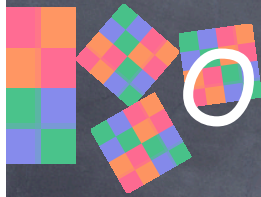
Negation:	- ~
Math:	^ * / + -
Equality:	= ~=
Identity:	== ~=
Comparison:	< > <= >=
Logical:	&
Assignment:	:=



64

This list is roughly in order of highest precedence to lowest.

- Unary Negation: - arithmetic negation (changes the sign of a number), ~ logical negation (changes true to false and false to true)
- Math: ^ power, * multiplication, / division, + addition, - subtraction
- Equality: = equal, ~= not equal; note that Dylan uses = for equality like Pascal, and unlike C/C++
- Identity: == two objects are the same object, ~= not the same object
- Comparison: < less than, > greater than, <= less than or equal, >= greater than or equal
- Logical: & logical and, | logical or; these only evaluate the right-hand side if the left-hand side is true or false, respectively, and return the value of the last expression evaluated
- Assignment: := is used for assignment, as in Pascal



Operator Functions

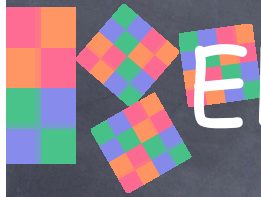
Infix Operator	Function Call
$1 + 2$	<code>\+(1, 2)</code>
$2 * 3$	<code>*(2, 3)</code>
$4 - 2$	<code>\-(4, 2)</code>
$-x$	<code>negative(x)</code>
$x = 42$	<code>\=(x, 42)</code>



65

Unary and binary infix operators are implemented using functions. You can refer to these functions by escaping the operator name with backslash. This allows you to call the function using function call syntax, or to pass the function around as any other function object.

Note that some operators are implemented in terms of others. For example, `\~=` is implemented by calling `\=` and inverting the result, and the magnitude comparison functions, like `\>=`, are implemented using `\<` and `\=`.



Element Reference

Element Reference

Function Call

`sequence[i]`

`element(sequence, i)`

`array[i, j, ...]`

`aref(array, i, j, ...)`

`all-windows[0]`

`element(all-windows, 0)`

`tic-tac-toe[1, 2]`

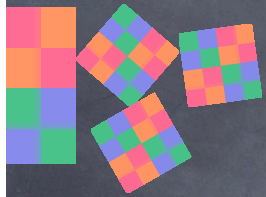
`aref(tic-tac-toe, 1, 2)`



66

Looking up elements of sequences and arrays, which are types of collections, is accomplished by calling **element()** for sequences and **aref()** for multidimensional arrays. The shorthand syntax **sequence[i]** and **array[i,j,k]** provide a convenient notation for calling these functions.

Calling the functions directly may provide access to additional function arguments, and you can implement these functions for collection classes you define. This means you can use the shorthand syntax for your classes just like the built-in ones.



Slot Reference

Slot Reference	Function Call
<code>argument.function</code>	<code>function(argument)</code>
<code>window.position</code>	<code>position(window)</code>
<code>window.view.origin</code>	<code>origin(view(window))</code>
<code>view(window).origin</code>	"
<code>origin(window.view)</code>	"



67

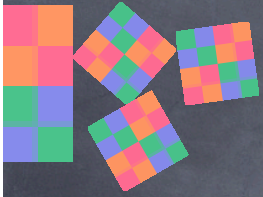
Dylan provides a shorthand syntax for calling functions that accept one argument. **argument.function** applies **function** to **argument**, just as if you had written **function(argument)**. This is commonly used to access object slots.

Slots are accessed using accessor functions, so you can write **slotname(object)** to get the value of the slot **slotname** for a given instance **object**. You can also use this shorthand syntax **object.slotname**, much as in C/C++ and Pascal. It is important to note that these are merely different syntax for the same thing—calling a function—quite unlike C/C++ and Pascal, where members/fields are accessed directly with the “dot” notation.

This is quite powerful. All access to objects is uniformly via functions, and it’s really just a matter of documentation or emphasis as to which ones are “slots” and which are not. The difference is less important to client code than it is in some other languages. This provides a level of encapsulation that makes it easier to change the implementation of a “slot” from simple access to something more complicated like lazy evaluation using an expensive computation, or performing other side effects.

You can also cascade slot reference syntax or even use either syntax in the same expression, as in the last three examples, all of which result in the same series of function calls.

There is one difference between these variations: The order of evaluation of the argument and function values is left-to-right according to the order in which they are written, not the order of the resulting function call.



Assignment

Variables

```
volume := 11;  
*total* := *total* + 1;
```

Functions

```
foo[i] := sqrt( x );  
window.height := 42;  
height( window ) := 42;
```



68

The assignment operator := is used to set variables to new values, and as an alternative to calling setter accessor functions, including **element-setter()** and **aref-setter()** to set elements in sequences and arrays.



Assignment (-setter)

Slot Reference

```
window.height := 42;  
height( window ) := 42;  
height-setter( 42, window );
```

Element Reference

```
foo[1] := 10;  
element( foo, 1 ) := 10;  
element-setter( 10, foo, 1);
```

Array Element Reference

```
bar[1, 2] := 11;  
aref( bar, 1, 2 ) := 11;  
aref-setter( 11, bar, 1, 2);
```

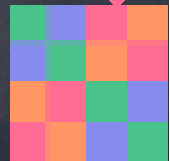
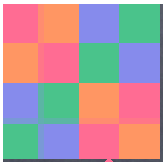


69

The assignment operator takes function names on the left-hand side and appends the suffix “-setter” to get the corresponding setter function, then calls it. It looks up the setter function name in the current lexical context (ie., the function names, including **element-setter** and **aref-setter** can be shadowed by local variables or imported from some other library, just like any other function).

In each of the groups above, each expression is equivalent (except for evaluation order of operands).

Multiple Values

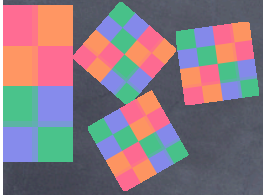




Multiple Values

- Functions can return multiple values, just like they can accept multiple arguments
- Eliminates the need for "output" parameters
- There is no "wrapper" object for the values; for example, on PowerPC, function arguments are stored in r3, r4, r5, etc. Multiple values could be returned in r3, r4, r5, etc.





values()

```
values( 1, 2, 3 );
```

```
⇒ 1  
   2  
   3
```

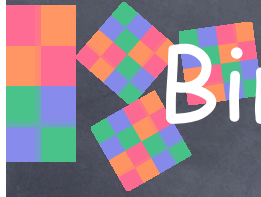
```
define method square-and-sum (x, y)  
  values( x ^ 2, x + y )  
end method;
```

```
square-and-sum( 2, 3 )
```

```
⇒ 4  
   5
```



You can return multiple values by calling the **values()** function, which takes its arguments and returns them as multiple values.



Binding Values (1 of 2)

```
let (a, b, c) = values( 1, 2, 3 );
```

```
a ⇒ 1
```

```
b ⇒ 2
```

```
c ⇒ 3
```

```
define variable (*whole*, *remainder*) =  
    truncate( 1.98 );
```

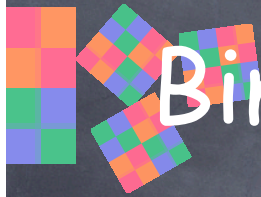
```
*whole*      ⇒ 1
```

```
*remainder* ⇒ 0.98
```



73

You can bind (assign) multiple values to local variables with **let** or to module constants and variables with **define constant** and **define variable** by using parenthesis.



Binding Values (2 of 2)

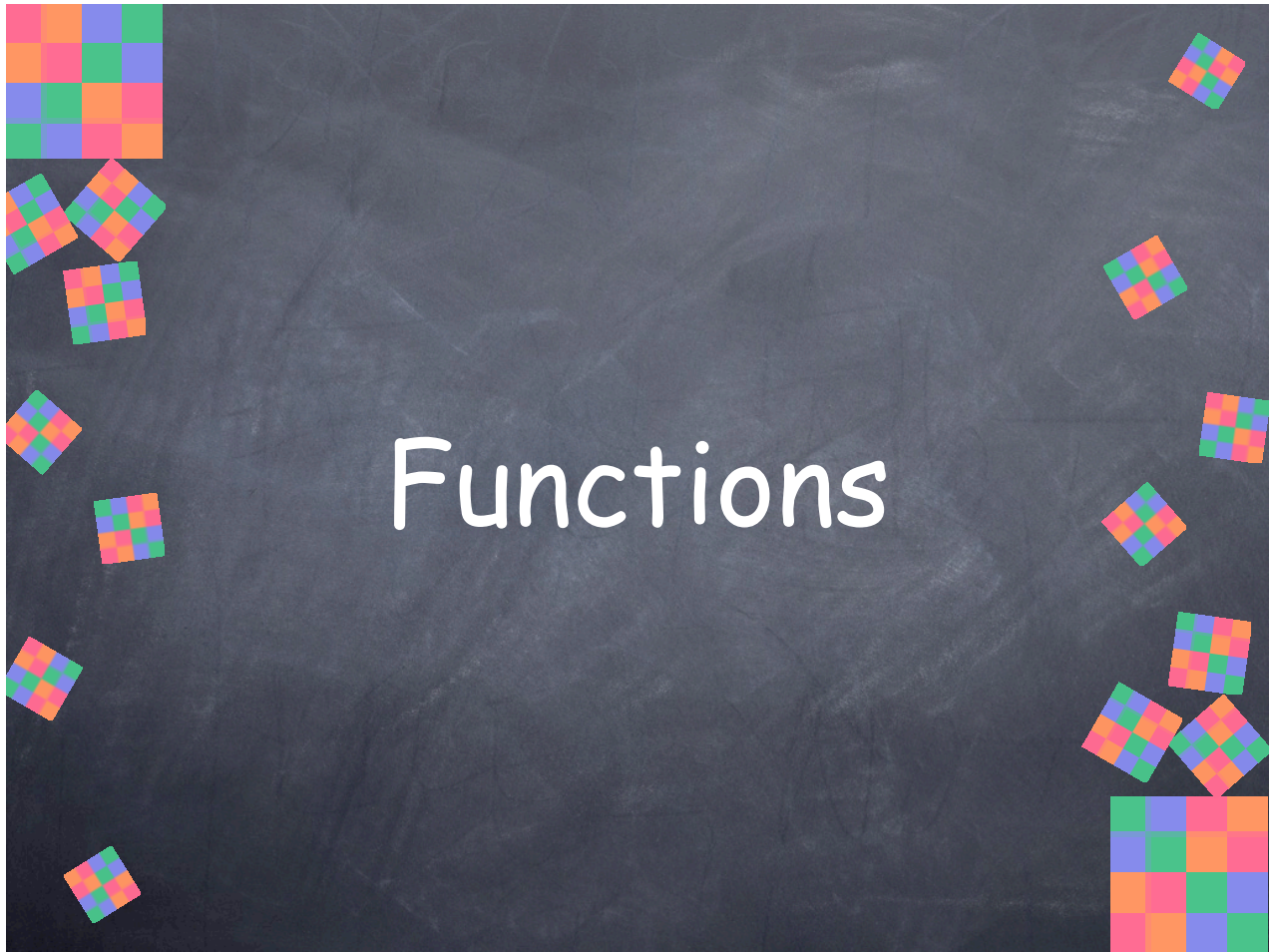
```
let (a, #rest r) = values( 1, 2, 3 );  
a ⇒ 1  
r ⇒ #(2, 3)
```

```
let (a, b, c) = values( 1, 2 );  
a ⇒ 1  
b ⇒ 2  
c ⇒ #f
```



74

- As in function signatures, **#rest** can be used to bind a list of any remaining values.
- If there are more bindings than values, then any remaining bindings are assigned **#f**.



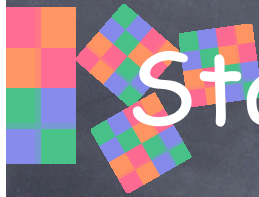
75

Dylan functions can have required arguments, `#rest` arguments, and keyword arguments with optional default values. They can return required and optional values. Methods must be congruent with their generic functions.

[SLIDES NOT YET WRITTEN]



An overview of the built-in statements.



Statements Overview

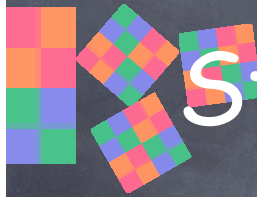
- Statements evaluate to a value and can be used anywhere an expression can, e.g.:

```
let x = if (foo) 42 else 0 end;
```

- Conditional statements treat any value other than #f as "true". Notably, zero and '\0' are not #f:

```
if (0) #"foo" else #"bar" end;  
⇒ #"foo"
```





Statements Syntax

- Statements optionally end with the "begin word", e.g.:

```
if...end if;  
for...end for;  
case...end case;
```

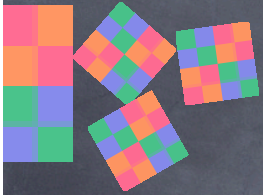
- Semicolons are optional before statement ends:

```
if (x) y; end;  
if (x) y end;
```



78

In Pascal, semicolons are statement separators, and you can't have one before an **end**. In C, semicolons are statement terminators, and you must have one after every statement. In Dylan, semicolons are optional before an **end**. A common idiom is to omit the semicolon to indicate that the value is being returned. In this idiom, a trailing semicolon indicates that the value is not returned or not used by the caller.



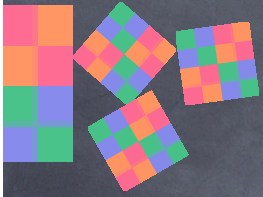
if

```
if (camel.humps = 1)
  "dromedary"
elseif (camel.humps = 2)
  "bactrian"
else
  "not a camel"
end if;
```



79

Dylan has a typical **if** statement. **elseif** is used to chain multiple tests without increasing nesting. The **elseif** and **else** clauses are optional, of course. If a test succeeds, the **if** statement returns the value of the matching “then” code, otherwise it evaluates to #f (false).



unless

```
unless (danger?( will-robinson ))  
  follow( dr-smith )  
end unless;
```



80

To complement **if**, Dylan has **unless**. It is equivalent to “if (~test) foo end”. It has no **else** clause. If the test evaluates to true (and “foo” isn’t executed), **unless** returns #f.



case

```
case
  camel.humps = 1 => "dromedary";
  camel.humps = 2 => "bactrian";
  otherwise      => "not a camel";
end case;
```



81

Case is like **if...elseif...elseif...else...end**; It evaluates each test in order and if a test evaluates to true, it executes the right-hand side of the arrow ("=>") and returns its value. An optional **otherwise** clause will match if no other test does. If there is no match, it returns #f.

- Unlike C, no explicit "break" is needed to end a right-hand side, and unlike Pascal, no explicit "begin/end" are needed. You can safely add as many expressions and statements as you like on the right-hand side.



select

```
select (camel.humps)
  3, 4, 5 => "mutant camel";
  1       => "dromedary";
  2       => "bactrian";
  otherwise => "not a camel";
end select;
```



82

select compares a value against a series of values, in order, and executes the right-hand side of the arrow if there's a match. More than one test value can be given on the left-hand side of each arrow. Like **case**, it has an optional **otherwise** clause. Notably, if there is no match (and no **otherwise**), an error is signaled.



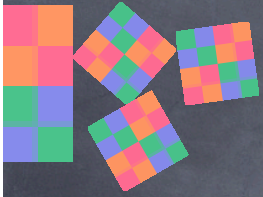
select (by)

```
select (my-object by instance?)  
  <window>, <view> => "UI object";  
  <number>, <string> => "computational";  
  otherwise => "unknown";  
end select;
```



83

By default, the **select** comparison is performed with “`\==`”. A different test function can be specified with **by**.



while

```
while (more-data?( stream ))  
  read-and-process-data( stream )  
end while;
```



84

while is an iteration statement. It loops as long as the test evaluates to true. **while** always returns #f.



until

```
until (end-of-file?( file ))  
  read-and-process-data( file )  
end while;
```



85

until is like **while** except that it loops as long as the test evaluates to false. Note that **until** evaluates the test first, just like **while**. Neither performs the test after the loop body like C's "do { } while ()".



for

```
for (tree in forest)
  look-at( tree )
end for;
```

```
for (i from 1 to 10) ...
for (j from 0 below 10,
     k from 10 above 0 by -1) ...
```

```
for (thing = first-thing then next(thing),
     until: done?(thing)) ...
```



86

Dylan's **for** loop is quite versatile. It supports three different kinds of iteration and an explicit termination test, and multiple iterations can be performed in parallel (the loop exits when any of the iterations ends or the explicit test matches).

- Collection iteration clause: **in** iterates over every element in a collection.
- Numeric iteration clause: **from** iterates by integral values. **to** is inclusive, **below** and **above** are exclusive (the iteration stops before reaching the bounding value). **by** optionally specifies how much to change the value each iteration; 1 is the default.
- Explicit Step iteration clause: = assigns an initial value and **then** is evaluated each time through the loop to determine the next value.
- Termination clauses: One of **until:** or **while:** can be given to supply an explicit termination test expression.

Taken together, an explicit step iteration and **while:** termination clause are like C/C++'s three for loop clauses.



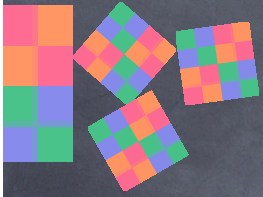
for (finally)

```
for (tree in forest,  
     count = 0 then count + 1,  
     while: have-film?( camera ))  
  photograph( tree );  
finally  
  print( photos );  
  count  
end for;
```



87

The **for** loop has an optional **finally** clause that can be used to perform some action after looping or to calculate a value for the statement to return. Iteration variables (except for collection iteration variables, since they may be invalid at the end of the loop) are available in the **finally** clause. If there is no **finally**, **for** returns #f.



begin

```
begin  
  do-stuff();  
  more-stuff();  
end;
```



88

Begin simply evaluates the expressions in order. It returns the result of the last expression. If there are no expressions, it returns #f.



block

```
block (return)
  open-files();
  if (files-empty?())
    return( #f );
  end;
  process-files();
afterwards
  report-totals();
cleanup
  close-files();
end block;
```



89


Block supports exception handling and non-local exits via an exit function. You can optionally supply a name in the parenthesis, and **block** will create a local variable with that name and bind it to a function that exits the block. "return" is just an example name, it isn't special like "return" in C. If any values are passed to the exit function, **block** returns them as its result. The exit function can be called anywhere in the block body, or even passed to other functions, which may call it to perform a non-local exit (exits any intervening functions on the stack, then exits the block—it's similar to C's `setjmp()/longjmp()`).

- The optional **afterwards** clause is executed after producing the value for the statement; this allows you to avoid having to create a local variable just to hold the value until the end of the block body.
- The optional **cleanup** clause is always executed whether or not an exception occurs or the exit function is called.
- If an exception occurs, the optional **exception** clauses are tested in order for one that can handle the exception and executes the first handler that can (if any).



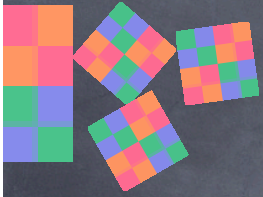
block (exception)

```
block ()  
  let result = do-stuff();  
  if (result ~= $no-error)  
    error( "Something's wrong!" );  
  end;  
  do-more-stuff();  
exception (e :: <file-error>)  
  print( e.file-name )  
exception (<error>)  
  beep()  
end block;
```



90

Some example **exception** clauses. **error()** signals an **<error>**. Exception clauses support other options, as well.



method

```
method (x)  
  x + 1  
end method;
```

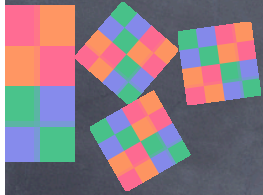


91

Method defines an anonymous function. It's like "lambda" in Lisp or Scheme (and several other languages), and `boost::lambda` in C++.



Local Declarations



let

```
let x = 0;  
let sym :: <symbol> = #"green";  
  
let (whole, rem) = truncate( amount );  
  
let (whole :: <integer>, rem :: <real>) =  
    truncate( amount );  
  
let (x, #rest rest) = values(1, 2, 3);  
x      ⇒ 1  
rest  ⇒ #(2, 3)
```



93

let defines a local variable and initializes it.

- An initial value is required—no variables may be left uninitialized.
- Local variables can be specialized—you can declare a type for them. Any attempt to assign a value of a different type will signal an error.
- **let** also supports multiple values. You can define a list of variables and assign a multiple-value value to them. You can also supply **#rest** to get a list of any remaining values. If there are more bindings than values, any remaining bindings are initialized to **#f**.



local

```
local method square (x)
  x * x
end method;
let y = square( 12 );
⇒ 144
```

```
local method back ()
  forth()
end,
method forth ()
  back()
end;
```



94

local defines one or more local methods.

- Multiple methods defined in the same **local** declaration are defined simultaneously and can be mutually-recursive.
- The word **method** is optional.



let handler

```
let handler <warning> =  
  method (warning, next-handler)  
    beep()  
  end method;
```



95

let handler establishes a condition handler that remains in effect until the end of scope. Exceptions are a particular kind of **<condition>**. **let handler** allows you to define a handler for any condition, not just errors. For example, you can use this to establish a **<restart>** handler to recover from errors.



Classes defined in the standard Dylan library.

[SLIDES NOT YET WRITTEN]

Types & Classes

[SLIDES NOT YET WRITTEN]



Characters, symbols, and booleans.

[SLIDES NOT YET WRITTEN]

Numbers

[SLIDES NOT YET WRITTEN]



100

The standard Dylan library includes a rich set of collection classes, including vectors, arrays, lists, hash tables, and strings.

[SLIDES NOT YET WRITTEN]

Functions

[SLIDES NOT YET WRITTEN]

Conditions

[SLIDES NOT YET WRITTEN]



Sources for more information about Dylan, implementations, libraries, and projects created with Dylan.



Implementations

- Gwydion Dylan, Gwydion Dylan Maintainers
 - <http://www.gwydiondylan.org/>
- Functional Developer, Functional Objects, Inc.
 - <http://www.functionalobjects.com/>
- Apple Dylan TR, Apple Computer, Inc.



104

- Gwydion Dylan is an open-source, portable implementation of Dylan for Mac OS X, Linux, and others. It consists of a command-line compiler, d2c, which generates C code, then compiles that, and the Mindy interpreter, and incomplete implementation used for bootstrapping d2c and interactive Dylan exploration.
- Functional Developer (aka “FunDev”) is a commercial implementation for Windows (95/NT/2000/XP) with a graphical IDE including an interactive listener and debugger. It is currently being ported to Linux and is in alpha testing (as of September, 2003). It was originally developed by Harlequin. I worked on the IDE at Harlequin for two years, programming entirely in Dylan.
- The Apple Dylan Technology Release may still be found if you look for it, although Apple no longer sells it, and I have found no online archive of it. There are screenshots and descriptions of it available online.



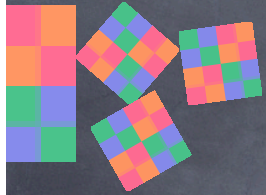
Projects

- The Monday Project: Libraries and tools for compiling, markup languages and text processing, in Dylan using XML-based literate programming
 - <http://monday.sourceforge.net/>
- Dylan Code Collection: Miscellaneous open-source Dylan libraries
 - <http://dylanlibs.sourceforge.net/>



105

- The Monday Project is an impressive example of Literate Programming at work, as well as a source for open-source Dylan libraries and tools.
- Dylan Code Collection includes useful examples like simple HTTP and SMTP servers.



Documentation

- The Dylan Reference Manual
 - <http://www.gwydiondylan.org/drm/>
- Dylan Programming: An Object Oriented and Dynamic Language
 - http://www.gwydiondylan.org/books/dpg/db_1.html



106

- The Dylan Reference Manual (aka “The DRM”) is the the official source for the language definition and the definition of the core “dylan” library. Some parts are dense, but a lot of it is quite readable—if you like reading language reference manuals, like I do. It is available online in HTML and PDF formats, as well as in print.

- Dylan Programming is an excellent tutorial that serves as an introduction to Dylan and object-oriented programming. It is available online in HTML format, as well as in print.

Miscellaneous other introductory material, tutorials, whitepapers and articles are also available on the Gwydion Dylan site.



Fire away!