# Course Notes: CPSC 4100 Spring 2020

## General Concepts

### Binding Times

The act of associating *names* with properties (data type, address, value) is called *binding*, and different properties are bound at different times.

```
#include <math.h>


  void main()
  {

  int i;
  double sum=0;

      for (i=1; i<100; i++)
           sum += sqrt(i);
  }
```

- **language definition time**

  meaning of keywords is bound – all implementations must behave the same way (void, for)

- **language implementation time**

  e.g. the range of values for `int` is implementation dependent. (not the same in java)

- **compile time**

  - data type for `i` is bound here. (static typing)
  - details of `sqrt` interface (declaration in math.h)

- **link time**

  definition of `sqrt`

- **load time**

  memory address for all of these symbols

- **runtime**

  `i` takes on a sequence of values

- early binding : before runtime / late binding == runtime binding
- not all language systems use all times (interpreters are not compiled)

## Parameter Passing Semantics

### Definitions

- formal parameters (specified in subroutine)
- actual parameters (passed to subroutine)
- the call stack

### parameter *correspondence*

- java and C use positional parameters
- other languages may have keyword parameters
- default parameters (C++ has this)
- variable arguments in C processed with system calls

### Call-by-value

- formal parameters are local variables in the stack frame (aka *activation record*) of the called method
- initialized with the value of the corresponding actual parameter
- variables used in calling function cannot be directly modified since only the values are passed (pointers & references complicate this)

### Call-by-reference

The lvalue of the actual parameter is computed before the method executes. Formal parameters are replaced with actual parameter's lvalue. Effectively, the formal parameters become aliases for the actual parameters.

### Call-by-macro-expansion

- formal parameters replaced with text of actual parameters
- macro call replaced with expanded macro

- variable capture For a given code snippet
  - **free variables** have no binding (are not associated with a specific memory location)
  - **bound variables** do
  - Macro expansion can cause free variables to become bound inside the macro expansion – this has undefined semantics and will result in errors (see the SWAP exmaple)

### Call-by-name

Formal parameters are substituted (in a capture-avoiding way) with "text" of actual parameters.

Also: **call-by-need** semantics: this is done in a memoized way (values are cached so it should be faster for certain applications)

## Parameter evaluation

- applicative order: parameters are evaluated before subroutine is called (this is typical for C, Java, scheme. . . )
- normal order: actual parameters are substituted into subroutine body and evaluated after subroutine call begins
  - see, Call-by-name

## Typing Systems

- rules surrounding the binding of data type to variables and expressions
- statically typed – data types bound at compile time
- type safety: how aggressively does the language apply typing rules to force you to write safe, good code
  - strongly typed <—> weakly typed
    C is (relatively) weakly typed because we can throw away type information by casting to `void*`

### Duck typing

Dynamic typing of objects. similar to informal interfaces.

Object variables have methods provided by type of content. This means that legal method syntax depends on the contents of the runtime value.

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck

### Definitions

#### statements vs. expressions

- a statement is a executable step in the algorithm
  - the building block of an algorithm
- an expression is anything with a value (can be evaluated)

#### lvalues and rvalues

Every expression is either an lvalue or rvalue

- rvalue's are temporary and have the lifetime of the corresponding expression
- lvalue's persist beyond the expression
  - variables (anything with a name)
- C has the ability to convert between the two in a manner

## Languages

### C

- originally 1973
- Dennis Ritchie (The R in K&R)
- ANSI (American National Standards Institute) C standard since 1989
- imperative: statements affect program state
- structured: formal control structures / blocks
- procedural: code organized into called procedures (subroutines)
- static typing: data type property assigned at compile time
- weakly typed (`void*`) : implicit type casting under some conditions
- compiles all the way to the hardware (executables not portable)
- allows for raw memory management and manipulation
- modeled naturally on the standard *von Neumann machine* architecture
  - CPU with registers, ALU, control unit
  - memory containing both instructions and data

**hello world in C (the parts of)**

```
#include <stdio.h>              /* preprocessor directive */

int main(int argc, char *argv[]) /* program entry point with command line arguments
{
    printf("Hello World\n");    /* a subroutine that does IO -- declared in stdio.h
    return 0;                   /* return code from main -> exit code for program */
}
```

**C data types**

```
 void
 char
 int
 float
 double

 // these assume int
 short
 long
 signed
 unsigned

// may be optimized by using read only memory
const

// no implementation-independent semantics
volatile
```

- typedef syntactic renaming of a type

  ```
  typedef unsigned int uint
  typedef const double CONSTANT;
  ```

- arrays

  ```
  int values[10];  -->   int* values
  ```

  array lookups as math problems

```
int values[10];

for (int i=0; i<10; i++)
{
    printf("%d: %d\n", i, values[i]);
}

printf("set values[3]\n");
/* type of values tells it how to interpret the 3 */
*(values + 3) = 999;

/* raw addresses */
printf("set values[8]\n");
*((int*)((void*)values + sizeof(int)*8))=1234;
```

## C++

- Bjarne Stroustrup
- first appeared 1985; standardized in 1998
- adds object oriented features, namespaces, generics, exceptions

### hello world in C++

```
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

### Null-terminated strings

String data is stored as an array of characters with the NULL character (ascii 0) indicating the end of the string.

Many useful functions for dealing with data of this type are available in `string.h`.

## Scheme

A lexically scoped dialect of Lisp

Strongly typed / dynamically typed.

### data types

- numbers: `1.0`, `45`, `8+3i`, ...
- characters: `#\x`
- booleans: `#t #f` (note this is different than `#\t` and `#\f`)
- symbols: 'foo
- strings: "hello"
- vectors: `#(1 2 3)` – like a fixed-length list

### procedures

- created with a `lambda` expression

```
(lambda (a b)
  (+ a b))                                ;add a and b
```

- bound to a variable with the `define` special form

```
(define add2 (lambda (a b)
               (+ a b)))
```

### function currying

concept from the lambda calculus where procedures may only take a single parameter.

f(a,b) = {f'(a)}(b)

For example, if lambdas could only take one argument:

```
(define add (lambda (a) (lambda (b) (+ a b))))
((add 5) 10)
```

**read from user**

- `(read)` -> symbol
- `(use-modules (ice-9 readline)) (readline "enter a string")` -> string

**control structures**

- if/else
  - eq? equal? ...
- when/unless
- loops do exist: while / do
- begin (implied within lambda)

**working with lists/pairs**

- cons
- car / cdr
- car, caar, cddr, cadr, etc.
- list / pair procedures
- memq , assoc

`cons` creates a new list with the first parameter stuck on the beginning of the second.

```
scheme@(guile-user)> (cons 'a '(1 2 3))
$2 = (a 1 2 3)
```

`car` returns the first item in a list. `cdr` returns the rest of the list, after the `car`

```
scheme@(guile-user)> (car '(this that the other))
$3 = this
scheme@(guile-user)> (cdr '(this that the other))
$4 = (that the other)
```

**recursion as iteration**

- factorial
- map, filter

**binding local variables introduce new variables**

- let, let*, letrec, letrec* let* -> nested lets
- nested define -> letrec

**closures**

A function with environment containing free variables bound in some other *environment* that existed when the function was created.

```
(define (make-adder base)
  (lambda (i)
    (+ base i)))
```

**imperatives**

- `set!` (and many other variants)

**lambda as object**

- closures and imperatives together give object-like functionality

## Scripting Languages

- often distinguished from *systems programming* languages like C that are ...
  - statically typed
  - compiled to machine code
- See Scripting: Higher-Level Programming for the 21st Century

The concept of *scripting* originally emphasized the development of *glue code* for connecting different software components (likely existing executables written in a systems programming language).

To facilitate use, these languages adopted fast development cycles avoiding separate compilation steps, and providing syntactic shortcuts like dynamic, weak typing.

Modern scripting languages are full-blown high-level languages in their own right, and may better be described as *dynamic languages* due to enhanced runtime capabilities such as

- dynamic typing
- first class functions (a dynamic form of subroutine construction)
- `eval`

## Perl

- A classic example of a scripting language run amok.
- Very popular in early web development (CGI Scripts).
- Fallen out of favor due to ugliness of code.
- Still extremely powerful for glue code.
- Unparalleled text processing capability.
- Language features:
  - imperative and functional aspects
  - garbage collection
  - objects
  - closures / first-class functions
  - native regular expressions
  - reflection
  - extensive library
  - call-by-reference parameter passing of scalars
- Tutorials: learn.perl.org
- Initially created by Larry Wall in 1987.

- A great example of a glue code language. It has some interesting capabilities including an amazing implementation of regular expressions, and has inspired many modern scripting languages.
  - closures / first-class functions
  - objects
  - reflection
  - garbage collection

**Syntax**

- Weak and dynamic typing
- Sigils are used to distinguish between structure of data
  - `$foo` is a scalar
  - `@foo` is a list, with elements accessed using square brackets: `$foo[0]` (notice that the content at index 0 is a scalar)
  - `%foo` is an associative array (aka a hash map, and often called a *hash*) with keys specified using curly braces: `$foo{"key"}`
  - `&foo` is a subroutine, though fortunately this particular sigil is often optional
  - `*foo` is a *typeglob* representing all the above at once (technically it corresponds to the symbol table entry for `foo`). In old-style perl, this is also used to represent filehandles. This can be used to set up symbol table aliases, and is not often needed.
- variables are global by default
- weird punctuation variables, such as `$_` the *default* variable
- parentheses are often optional
- References are scalars created using backslash. They are dereferenced using the desired sigil of the target value.

```
$myref = \%somehash;

# one way to access the refrerenced hash:  %$myref
```

- Subroutines
  - All parameters are passed as one list, accessed within the subroutine using the special variable `@_`
  - Scalar parameters are passed by reference
  - lexically scoped variables are declared with `my`
- Notable aspects

- objects offer no encapsulation (at least in Perl 5)
- lexical and dynamic scope, depending on how variables are declared
  Only globals and locals show up in the symbol table, but we can examine the symbol table directly perl's dynamic scoped local variables give us a way to effectively temporarily override global variables; Here's a classic perl oddity:

```
{
  local $"=';';
  print "@b"
}
```

Local variables create a temporary symbol table that goes away when the variable leaves scope.

### Python

- Object-oriented
- dynamic, strong typing, duck
- has REPL
- elements of functional
- supports performance can be converted to C using Cython
- exceptions
- closures
- Special values: True, False, None
- simple, whitespace-sensitive syntax
- Very popular with non-programmers. Modular library, data science, semantic analysis , high performance computing, domain programming

"Python's philosophy rejects the Perl "there is more than one way to do it" approach to language design in favor of "there should be one—and preferably only one—obvious way to do it"."

- Duck typing

**idioms**

- tuple packing/unpacking

```
b,a = a,b
```

- list comprehensions

```
nums = [ n*2 for n in range(10) if n%2]
```

- generators **lazy list comprehensions**

```
nums = (n*2 for n in xrange(10))
```

- for / else
- type coercion – e.g, you often need strings, so convert `str(n)` – different than casting!

## Erlang

Agner Erlang was a Danish mathematician (1878-1929) that founded the mathematical analysis of telecommunication networks.

It's also a portmanteau of "Ericsson Language"

Developed originally in 1986, for telephone switches, open sourced in 1998. And enjoys a recent resurgence in popularity. (facebook chat > 200 million active users; ejabberd)

http://www.erlang.org/doc/

http://www.tryerlang.org

http://learnyousomeerlang.com

**Notable aspects of Language**

- a benchmark with 20 million processes has been successfully performed.
- Ericsson switch built with 1million+ lines of erlang, "9 9's reliability" – 31ms downtime /yr.
- functional language and runtime system
- byte compiled
- garbage collected
- strict evaluation (eager / immediate / not lazy)
- single assignment (weird!) – variables don't vary: referential integrity
- dynamic, strong typing
- hot swapping
- fault-tolerant: "Let it Crash"
- based on pattern matching
- has a REPL
- tail call optimized
- concurrency through explicit message passing
  - erlang processes are very lightweight
  - use little memory (300 words per)
  - construct and destroy in microseconds
  - handled by the VM
- distributed address space

**Syntax**

- numbers
  - specify the base 2#10110
  - other syntax for dealing with binary data / patterns
  - atoms (symbols) start with lowercase letter or are in single quotes
  - true / false atoms
  - andalso / orelse – shortcut
  - variables start with Uppercase letter
  - tuples in curly braces
  - lists in square
    - `hd`, `tl` – like cons, cdr
    - append with `++` ; remove with `--`

  - pattern matching used for assignment
    * `[Head|Tail]=mylist`
    * `{Hourbinary, Minute, Second}=time().`
    * The "I don't care" variable: `_`

- functions provided in modules specified with colons
  - functions have different **arity**: the required number of parameters
- functions use arrows
- functions can be anonymous `fun(X) -> X end.`
- map, filter, fold (like a reduce)
- exceptions: throw() try/catch

### Prolog

- Logic programming language originally appearing in 1972

- declarative
- based in formal logic
- GNU prolog
- rule-based logical queries / "expert systems"
- http://www.learnprolognow.org

- gprolog
  - compiles to C
  - has a REPL
  - constraint solving over finite domains
  - runs against its loaded list of predicates

- syntax
  - facts, rules and queries
  - statements end with `.`
  - exit gprolog with halt.
  - variables
    * start with capital letter
    * `_` is the anonymous variable
    * a placeholder not unified to any specific term

  - numbers
  - atoms
    - lowercase or quoted
  - structure
    - tuples of terms tagged by a functor (atom syntax) with `arity`
  - lists, denoted by `[]`
  - rules denoted by `:-`

- variable unification occurs when prolog runtime discovers a value for a variable that leads to truth
- arithmetic with is instead of =