CT331

PROGRAMMING PARADIGMS

λ

Andreas Ó hAoḋa
University of Galway
2023-11-10

Contents

1 Introduction

1.1 Lecturer Contact Information

- Finlay Smith, School of Computer Science.
- `finlay.smith@universityofgalway.ie`

1.2 Syllabus

This module introduces three different programming paradigms: Procedural, Functional & Logical. This will involve 3 programming languages: C (mostly function pointers - knowledge of C is assumed), LISP (a functional language) and Prolog (a logical language). Both LISP and Prolog will both be introduced but neither will be fully covered in this module. There are no books required or recommended for this course.

1.2.1 Marking

30% of the marks for this module will be for the three assignments (one for each paradigm). The remaining 70% of the marks will be for the written exam.

1.3 Programming Paradigms

A **paradigm** is a typical example or pattern of something; a pattern or model. A **programming paradigm** is a pattern or model of programming. Various types of programming languages are better suited to solving particular problems. Programming language implementations differ on semantics & syntax:

- **Syntax** refers to the rules of the language; it allows us to form valid expressions & statements.
- **Semantics** refers to the meaning of those expressions & statements.

Programming languages can be classified according to the features that they have with respect to both the conceptual & implementation level. An alternative definition for a **programming paradigm** is a collection of abstract features that categorise a group of languages.
"*The popularity of a paradigm is due to one community deciding which problems are important to solve and then supporting the most promising paradigm for attacking these problems.*" – Thomas Kuhn.

1.4 Influences on Paradigms

- Computer Capabilities.
- Applications.
- Programming Methods: Language designs have evolved to reflect changing understanding of good methods for writing large & complex programs.
- Implementation Methods: Early compilers to optimised compilers; structured engineering to software engineering; data abstraction to OO.
- Theoretical Studies: Formal maths methods have deepened our understanding of strengths & weaknesses of language features and thus influenced the choice & inclusion of those features.
- Standardisation (has proved to be a strong conservative influence on the evolution of programming language design).

1.5 Why Learn Different Paradigms?

- Different paradigms make different trade-offs; What's tricky in one paradigm is "baked in" in another.
- Changing paradigms forces you to "change gears".
- It will prepare you for learning languages that you've never heard of or that may not exist yet.
- Helps you to decide what the best tool for the job is.
- Helps you to understand languages at a deeper level.

1.5.1 Why Learn Functional Programming?

- It's one of the oldest paradigm (Lisp: 1958, still widely used today).
- Heavily based on mathematical concepts (proofs, lambda calculations).
- Elegant solutions (recursion).
- Other paradigms can be interpreted in terms of functional programming.

1.5.2 Why Learn Logical Programming?

- Long history.
- ALlows implementation of things that are difficult in other paradigms.
- *Very* different
- Helps to conceptualise logical problems.

1.5.3 Why Learn Imperative Programming?

- The oldest paradigm – goes back as far as punch cards & magnetic loops.
- Much closer representation of how the machine actually works, i.e. "closer to the metal".
- Can help to recognise optimisation issues / computational bottlenecks.
- Contextualises many other systems (UNIX, Linux, etc.).

1.5.4 Why Learn Object-Oriented Programming?

- Tries to represent the real world.
- Abstraction & inheritance.
- Object-Oriented is everywhere.

2 Overview of Object-Oriented Programming

Object-Oriented languages include:

- Java.
- C#.
- VB.NET.
- Scala.
- JavaScript.
- Python.
- PHP.
- Smalltalk.
- Ruby.

2.1 Fundamentals of Object-Oriented Programming

- Everything is an object.
- Computation is performed by message-passing.
- Every **object** is an **instance** of a **class** which is a grouping of similar objects.
- **Inheritance** describes the relationships between classes.

Object-Oriented Programming focuses on the objects that the program represents and allows them to exhibit "behaviour".

2.2 Four Major Principles of OOP

2.2.1 Encapsulation

Data is hidden as if **encapsulated** within the object. Direct access to the data is restricted, instead we use methods to get, set, & manipulate data. Manipulation of data is hidden. The object caller doesn't need to know what's actually going on behind the scenes. We can be (fairly) sure that nobody else is fiddling with our data.

2.2.2 Abstraction

Functionality can be defined without actually being implemented. High-level interfaces provide method types/names without implementation. This allows case-specific implementation, allows one person to define the functionality & another to implement, and allows the representation to be changed without affecting "public" view of the class. This is helpful when designing large systems.

2.2.3 Inheritance

Classes can **inherit** functionality without re-implementing. This prevents the duplication of code. This is also helpful when designing large systems; it encourages a well-structured codebase.

2.2.4 Polymorphism

Objects of one class can be treated like objects of other classes.

3 Imperative & Procedural Programming

3.1 Imperative Programming

Imperative programming involves telling the computer to perform a set of actions, one after the other. Most programming languages have imperative aspects. Imperative programming consists of a list of instructions, `GOTO` statements, and little or no structure. E.g., Assembly.

3.2 Procedural Progamming

Procedural progamming splits actions into **procedures** or tasks. Procedures can be made up of other procedures (composition, recursion). The code is structured, uses "functions" or procedures, encourages code re-use, and encourages encapsulation & composition. Note that procedural functions are not to be confused with Functional Programming.

3.2.1 Structured Programming

Examples of structured programming languages include basically everything except Assembly.

- Code is structured.
- `while`, `for`, `if`, `else`, `switch`, `class`, `function`, etc.

- Less emphasis on `GOTO` statements.
- Creating a structure to manage instructions.
- Allows more complex programs to be built.
- Easier to understand.
- Helps to avoid `GOTO` bugs & spaghetti code.

3.3 The C Programming Language

C is a procedural, imperative, structured "systems language". It came into being around 1969-1973 in parallel with the development of the UNIX operating system. Basic Compiled Programming Language (BCPL) →B →C... C has had an ANSI standard since the 1980s. Now one of the most popular & powerful languages in use today.

```c
// header inclusion: functionally defined in stdio.h is added into the program by the compiler,
↪  specifically the Linker step of the compiler
// "stdio" is short for "Standard Input / Output"
#include <stdio.h>

// function prototype: tells the compiler that the function exists before it has been implemented
// allows the compiler to handle recursion, or functions calling each other
void sayHello();

// function definition: implements the function
// note: data type, arguments, return
void sayHello() {
    // calling a function: printf takes a char* argument
    printf("Hello World!\n");
}

// main function: the entry point to the progam
// returns int
// takes two arguments: argc (the number of command-line arguments) & argv (an array of the
↪  arguments)
int main(int argc, char* argv[]) {
    // calling a function: sayhello takes no argument. nothing is returned
    sayHello();
    return 0;
}
```

Listing 1: Example C Program: `helloWorld.c`

```c
#include <stdio.h>

int add(int a, int b);

int add(int a, int b) {
    return a+b;
}

int main(int argc, char* argv[]) {
    printf("Let's add some numbers...\n");
    int first = 8;
    int second = 4;
    printf("The first number is %d\n", first);
    printf("The second number is %d\n", second);

    // "add" is a function that returns an int
    // the returned int is stored in the "result" variable - they must have the same data type
    int result = add(first, second);


    // "%d" is for ints - strictly decimal ints
    // "%i" is any int including octal and hexadecimal
    printf("When we add them together we get: %d\n", result);

    return 0;
}

```

Listing 2: Example C Program: `addNumbers.c`

3.3.1 Pointers

```
int* p; // variable p is a pointer to an integer value
int i;  // integer value
```

You can **dereference** a pointer into a value with `*`.

```
// ineger i2 is assigned the integer value that the pointer p is pointing to
int i2 = *p;
```

You can get a pointer to a value with &.

```
// pointer p2 points to the address of integer i
int* p2 = &i;
```

A function effectively breaking the convention that arguments are not changed in a function is a **side effect**. This is done by passing addresses.

```
#include<stdio.h>

void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(int argc, char* arv[]) {
    int a = 8;
    int b = 4;
    swap(&a, &b);   // this should make a=4 & b=8
}
```

3.3.2 Arrays & Pointers

```c
int intArr[5];  // an integer array of size 5
// intArr is a pointer to the 0th element of the array - the same as &intArr[0]

intArr[2] = 3;  // same as *(intArr+2) = 3;
// (intArr + 2) is of type (int*) while intArr[2] is of type int
// in the latter case, the pointer is dereferenced
// (intArr + 2) is the same as (&(intArr[2]))
// note that the + operator here is not simple addition - it moves the pointer by the size of the
↪  type
```

3.3.3 Generic Swap function?

What about a swap function that works on any data type?

```c
void swap(void* x, void* y) {
    void temp = *x; // won't work!
    // we don't know what size data *x points to, so void temp can't work
    // it is impossible to have a variable of type void for this reason
    // but we can have a pointer of type void*

    *x = *y;
    *y = temp;
}
```

Listing 3: (Non-functional) Attempt at a Generic `swap` Function

`void*` is a specific pointer type which points to some location in memory. It has no specific type and therefore no specific size.

`sizeof(<type>)` returns the size in bytes of the object representation of `<type>`. `sizeof()` is built-in to the C language.

`void* memcpy(void* to, const void* from, size_t size)`. The `memcpy()` function copies `size` number of bytes from the object beginning at location `from` into the object beginning at location `to`. The value returned by `memcpy()` is the value of `to`. The `memcpy()` function is defined in `string.h`.

```c
#include <string.h>

void generic_swap(void* vp1, void* vp2, int size) {
    char temp_buff[size];   // need malloc?
    memcpy(temp_buff, vp1, size);
    memcpy(vp1, vp2, size);
    memcpy(vp2, temp_buff, size);
}
```

Listing 4: Generic Swap Function

3.4 Stacks vs Heaps

A **stack** is a LIFO data structure of limited size & limited access. It supports only two operations: PUSH & POP. Stacks are very fast. The limited size of stacks can result in stack overflow, and you cannot free memory in a stack except by POPping. To continue the `swap()` function from above:

```c
// first, a, b, & c are pushed onto the stack
char a = 'a';
int b = 100;
int c = 50;

// when swap() is called, x, y, & temp are pushed onto the stack
void swap(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;

    // when swap returns, x, y, & temp are popped from the stack and their memory is no longer in
    ↪  use
}

swap(b, c);
```

But what if we want to keep track of `temp` and use it later?

A **heap** is an unordered data structure of (theoretically) unlimited size and global access. The heap operations are allocate & free. Heaps are slower than stacks. Heaps are also harder to manage than stacks as they can get memory leaks.

```c
// first, b & c are pushed onto the stack
int b = 100;
int c = 50;

// when swap() is called, x, y, & temp are pushed onto the stack
void* swap(int* x, int* y) {
    int temp = *x;

    // we allocate space in memory to perm using malloc
    int* perm = malloc(int);
    perm = &temp;
    x = y;
    y = *perm;

    // when swap returns, x, y, & temp are popped from the stack
    // the memory allocated to perm is still in use
    return perm;
}


void* p = swap(b, c);
free(p);
```

Why not just return `temp` in the same way?

- Even when this function terminates, another function can access perm using that pointer.
- If we need to store a large or undeterminable amount of data, we can safely use the heap as there is no risk of stack overflow and no risk of losing reference or accidental de-allocation of memory.

4 Dynamic Memory

FINISH OFF

5 Functional Programming

Given the same problem to solve, a program for said problem in any programming language can be considered equivalent to any other at the machine level in that the programs will result in changes to values contained in memory cells. However, there can be quite significant differences at both the conceptual & implementation level.

5.1 Lisp, Racket, & Scheme

LISP (more commonly referred to as **Lisp**) is a contraction of **List Processing**. **Scheme** is a dialect of Lisp, and **Racket** is an implementation of Scheme. Lisp uses prefix (Polish) notation, e.g.: `(+ 3 4)`, `(* 5 6)`, `(- 4 (* 5 6))`, etc.

5.1.1 Function vs Literal

Parentheses are used to represent a **function**:

```
(+ 3 4)         ; =   7
(* 5 6)         ; =  30
(- 4 (* 5 6))   ; = -26
```

A single quote is used to represent a **literal**:

```
(+ 3 4)     ; = 7
'(+ 3 4)    ; = '(+ 3 4)
```

Rather than considering + as the name of a function, the quote means to take everything literally, i.e. "+" is just a word. Nothing is evaluated.

5.1.2 S-Expressions

Both code & data are structured as nested lists in Lisp. **Symbolic Expressions** or s-expressions, sexprs, or sexps are a notation for nested list structures. They are defined with a very simple recursive grammar, but produce a very flexible framework for computing. An s-expression is defined as:

1. An **atom**. Atoms are considered to be "indivisible". Primitive data types like numbers, strings, booleans, etc. are atoms. Lists & pairs (s-expressions) are not.

2. An expression in the form `(X . Y)`, where `X` & `Y` are s-expressions.

A **pair** is two pieces of data together. They are created by the `cons` function, which is short for "construct", e.g. `(cons 1 2)`. The two values joined with `cons` are printed between parentheses interspaced by a `.` (a period):

```
> (cons "banana" "split")
'("banana" . "split")'
```

5.2 Lists

A **list** is an ordered group of data. List elements are separated by a space. The list syntax is a shorthand for an s-expression. Lists are displayed between parentheses using the `'` (single quote character).

```
'(1 2 3)                       ; list of numbers
'("this" "that" "the other")   ; list of strings
'(1 2 "three" 4)               ; list of mixed data types
```

Lisp uses nested lists (which are essentially linked lists). We can access the first element of a list using the `car` function:

```
> (car '(1 2 3))
1
```

We can access the rest of the list using the `cdr` function:

```
> (cdr '(1 2 3))
'(2 3)
```

`cdr` is analogous to `element->rest` in our C linked list.

```
> (car (cdr '(1 2 3)))
2
```

There is a shorthand for a combination of `car`s & `cdr`s (up to 4 operations usually but it depends on the Scheme environment), where `*` is `a` or `d` or a combination (if supported). For example, write a sequence of `car`s & `cdr`s to extract: "`d`" from list `(a b c d e f)` named `lis`:

```
;;;; these two are equivalent
(car (cdr (cdr (cdr  lis))))
cadddr(lis)
```

Lists are really just `cons` pairs where these second element is another list or `empty`; `empty` is a special word, similar to `NULL` in other languages.

```
> (cons 2 empty)
'(2)

> (cons 1 (cons 2 empty))
'(1 2)
```

The built-in functions `list` & `append` provide a more convenient way to create lists.

5.2.1 `define`

`define` binds a variable to some data. Its format is `(define variable value)`. `define` is used for user-defined functions. Note that user-defined functions can be used within other use defined functions as long as the functions are defined before they are invoked.

```
(define (function_name parameter-list)
    Function-body
)

;;; calculates the absolute addition of two numbers where the function abs returns the absolute
↪  value of a number
(define (sumabs num1 num2)
    (+ (abs num1) (abs num2))
)
```

```
> (sumabs 2 -3)
5
```

5.2.2 `list` & `append`

The `list` function constructs a list from components. It takes the form `(list el-1 el-2 el-n)`. These components can be symbols, numbers, or lists.

`append` collects components from several lists into one list. Its arguments must be lists. `append` takes the form `(append list1 list2 listn)`.