

Computing Unit 1: R Basics



Kurt Hornik

September 26, 2018



Overview

- Definitions
- Programming Languages
- S and R

Definitions

Programming

Definitions

Programming To design, write, and test programs.

Definitions

Programming To design, write, and test programs.
(Computer) Program

Definitions

Programming To design, write, and test programs.
(Computer) Program A computer algorithm.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language System of symbols used for communication (information exchange). Consists of Syntax and Semantics.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language System of symbols used for communication (information exchange). Consists of Syntax and Semantics.

Syntax

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language System of symbols used for communication (information exchange). Consists of Syntax and Semantics.

Syntax The structure of strings in some language.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language System of symbols used for communication (information exchange). Consists of Syntax and Semantics.

Syntax The structure of strings in some language.

Grammar A formal definition of the syntactic structure of a language.

Definitions

Programming To design, write, and test programs.

(Computer) Program A computer algorithm.

Algorithm A detailed sequence of instructions (actions) used to do a particular job or solve a given problem.

Programming language An artificial language that is used to generate or to express computer programs.

Language System of symbols used for communication (information exchange). Consists of Syntax and Semantics.

Syntax The structure of strings in some language.

Grammar A formal definition of the syntactic structure of a language.

Semantics Meaning of a language (relation to the real world).

Imperative PLs:

- Implicit state: variables
- State modification: through assignment (“side effecting”)
- Instruction sequencing (begin-end blocks, loops, . . .)

Imperative PLs:

- Implicit state: variables
- State modification: through assignment (“side effecting”)
- Instruction sequencing (begin-end blocks, loops, . . .)

Declarative PLs:

- No implicit state, no assignments
- Expression evaluation instead of instruction sequencing
- Recursion instead of loops

First Generation Languages (1GL)

Languages of the first computer systems (1940s).

Raw machine code, i.e. numeric (binary) values interpreted as commands by the processor.

First Generation Languages (1GL)

Languages of the first computer systems (1940s).

Raw machine code, i.e. numeric (binary) values interpreted as commands by the processor.

Example: 00011010 0011 0100 (3 + 4)

First Generation Languages (1GL)

Languages of the first computer systems (1940s).

Raw machine code, i.e. numeric (binary) values interpreted as commands by the processor.

Example: 00011010 0011 0100 (3 + 4)

Second Generation Languages (2GL)

Aka assembler languages (early 1950s).

Symbolic representation of machine code. Use of macros (placeholder for a sequence of commands) is common.

First Generation Languages (1GL)

Languages of the first computer systems (1940s).

Raw machine code, i.e. numeric (binary) values interpreted as commands by the processor.

Example: 00011010 0011 0100 (3 + 4)

Second Generation Languages (2GL)

Aka assembler languages (early 1950s).

Symbolic representation of machine code. Use of macros (placeholder for a sequence of commands) is common.

Example: ADD 3,4

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)
- System independent (core functionality)

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)
- System independent (core functionality)
- Provide named variables

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)
- System independent (core functionality)
- Provide named variables
- Provide structure elements (loops, conditions)

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)
- System independent (core functionality)
- Provide named variables
- Provide structure elements (loops, conditions)
- Every 3GL program must be translated into machine code prior to execution, either command by command *during* execution (interpreter) or as a whole *before* execution (compiler).

Third Generation Languages (3GL)

High level languages. Key characteristics:

- Easy to understand (compared to assembler)
- System independent (core functionality)
- Provide named variables
- Provide structure elements (loops, conditions)
- Every 3GL program must be translated into machine code prior to execution, either command by command *during* execution (interpreter) or as a whole *before* execution (compiler).
- The 3GL program is called **source code**, the resulting machine code **object code**.

History of 3GLs

- 1950: COBOL (COmmon Business Oriented Language)
- 1955: FORTRAN (FORmula TRANslator)
- 1960: BASIC (Beginners All-purpose Symbolic Instruction Code)
- 1970: PASCAL, MODULA (Niklaus Wirth), C
- 1980: C++, Objective Pascal

Fourth Generation Languages (4GL)

- “Application specific” high-level languages, mostly built around database systems (late 1970s).

Fourth Generation Languages (4GL)

- “Application specific” high-level languages, mostly built around database systems (late 1970s).
- Powerful set of functions/commands, but slower execution than 3GL. Often vendor-dependent.

Fourth Generation Languages (4GL)

- “Application specific” high-level languages, mostly built around database systems (late 1970s).
- Powerful set of functions/commands, but slower execution than 3GL. Often vendor-dependent.
- Examples:
 - Query languages for interactive data retrieval (e.g., SQL)
 - Report generators
 - Graphics languages (e.g., PostScript)
 - Application generators, CASE tools (e.g., Delphi)
 - Very high-level programming languages (e.g., MATLAB, SAS)

Object-Oriented Programming Languages

- Objects model real-world entities.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.
- The *behavior* of an object is defined by its repertoire of methods.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.
- The *behavior* of an object is defined by its repertoire of methods.
- The *state* of an object is defined by its attributes (variables). Attributes are accessed through methods.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.
- The *behavior* of an object is defined by its repertoire of methods.
- The *state* of an object is defined by its attributes (variables). Attributes are accessed through methods.
- Objects are instances of *classes* (object templates).

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.
- The *behavior* of an object is defined by its repertoire of methods.
- The *state* of an object is defined by its attributes (variables). Attributes are accessed through methods.
- Objects are instances of *classes* (object templates).
- Classes can be hierarchically organized through inheritance of both methods and attributes.

Object-Oriented Programming Languages

- Objects model real-world entities.
- Each object is composed of data and code which are “encapsulated” from the other objects.
- An object is characterized through *state* and *behavior*.
- The *behavior* of an object is defined by its repertoire of methods.
- The *state* of an object is defined by its attributes (variables). Attributes are accessed through methods.
- Objects are instances of *classes* (object templates).
- Classes can be hierarchically organized through inheritance of both methods and attributes.
- Examples: C++, Java

Functional Programming Languages

- Computation based on function evaluation.

Functional Programming Languages

- Computation based on function evaluation.
- Ideally, no assignments (“side-effects”).

Functional Programming Languages

- Computation based on function evaluation.
- Ideally, no assignments (“side-effects”).
- Referential transparency: meaning of the whole is solely determined by the meaning of the parts.

Functional Programming Languages

- Computation based on function evaluation.
- Ideally, no assignments (“side-effects”).
- Referential transparency: meaning of the whole is solely determined by the meaning of the parts.
- Functions are first-class objects (treated like values)

Functional Programming Languages

- Computation based on function evaluation.
- Ideally, no assignments (“side-effects”).
- Referential transparency: meaning of the whole is solely determined by the meaning of the parts.
- Functions are first-class objects (treated like values)
- Lazy evaluation: expressions are evaluated only when needed

Functional Programming Languages

- Computation based on function evaluation.
- Ideally, no assignments (“side-effects”).
- Referential transparency: meaning of the whole is solely determined by the meaning of the parts.
- Functions are first-class objects (treated like values)
- Lazy evaluation: expressions are evaluated only when needed
- Examples: LISP, APL, S

Logic Programming Languages

- based on rules of formal logic

Logic Programming Languages

- based on rules of formal logic
- results are derived from rules

Logic Programming Languages

- based on rules of formal logic
- results are derived from rules
- base concept: unification

Two terms to be unified are compared. Both constants: result is TRUE or FALSE. One constant, one variable: variable is bound to constant. Two expressions: unified recursively.

Logic Programming Languages

- based on rules of formal logic
- results are derived from rules
- base concept: unification

Two terms to be unified are compared. Both constants: result is TRUE or FALSE. One constant, one variable: variable is bound to constant. Two expressions: unified recursively.

- Example: PROLOG (PROgramming with LOGic)

Examples: 1, 2, 3, ..., 10

COBOL

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. DisplayNumbers.
```

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 I PIC 99 VALUE 1.
```

```
PROCEDURE DIVISION.  
Begin.  
    PERFORM UNTIL I = 11  
        DISPLAY I  
        ADD 1 TO I  
    END-PERFORM  
STOP RUN.
```

Examples: 1, 2, 3, ..., 10

FORTRAN

```
PROGRAM DisplayNumbers  
  
INTEGER :: i  
  
DO 99 i = 1, 10  
    PRINT *, i  
99 CONTINUE  
  
END PROGRAM
```

Examples: 1, 2, 3, ..., 10

BASIC

```
10 FOR i = 1 TO 10  
20 PRINT i  
30 NEXT i
```

Examples: 1, 2, 3, ..., 10

PASCAL

```
Program DisplayIntegers;  
Var i : Integer;  
Begin  
  For i := 1 to 10 do  
    WriteLn(i);  
End.
```

Examples: 1, 2, 3, ..., 10

C

```
void main() {  
    for (int i = 1; i < 10; i++)  
        printf("%u\n", i);  
}
```

Examples: 1, 2, 3, ..., 10

LISP

```
(dotimes (i 10)  
  (print (+ 1 i))  
)
```


Examples: 1, 2, 3, ..., 10

S

```
print(1 : 10)
```