Philology of Programming Languages

Baptiste Mélès

Université de Clermont-Ferrand, PHIER / Archives Poincaré

History and Philosophy of Computing (HaPoC),
École Normale Supérieure de Paris,
30th October 2013

Theoretical *vs.* real-life programming languages

Do "theoretical" programming languages capture all the properties of "real-life" programming languages?

- theoretical programming languages:
  - defined in papers;
  - used to prove theorems of logic or computer science;
  - examples: all sorts of λ-calculi, Turing machines, PCF...
- real-life programming languages:
  - implemented on computers;
  - used to write real and useful programs (operating systems, browsers, shells...);
  - examples: Lisp, C, Perl, C++, JavaScript...

Theoretical *vs.* real-life programming languages

Do "theoretical" programming languages capture all the properties of "real-life" programming languages?

- theoretical programming languages:
  - defined in papers;
  - used to prove theorems of logic or computer science;
  - examples: all sorts of λ-calculi, Turing machines, PCF...
- real-life programming languages:
  - implemented on computers;
  - used to write real and useful programs (operating systems, browsers, shells...);
  - examples: Lisp, C, Perl, C++, JavaScript...

Theoretical *vs.* real-life programming languages

Do "theoretical" programming languages capture all the properties of "real-life" programming languages?

- theoretical programming languages:
  - defined in papers;
  - used to prove theorems of logic or computer science;
  - examples: all sorts of λ-calculi, Turing machines, PCF...
- real-life programming languages:
  - implemented on computers;
  - used to write real and useful programs (operating systems, browsers, shells...);
  - examples: Lisp, C, Perl, C++, JavaScript...

The compilation fallacy

- The compilation fallacy: all programming languages are "equivalent", since most of them are
  - Turing-complete
  - compilable into binary/assembly/your favourite language;
- Felleisen 1990: "Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal".
- But:
  - no programmer writes real programs in assembly language or Turing machines;
  - compilation collapses all specific properties of programming languages, which every programmer feels in his practice.
- Thus, *programs must not be examined up to compilation*, but as source codes, before their compilation.

The compilation fallacy

- The compilation fallacy: all programming languages are "equivalent", since most of them are
  - Turing-complete
  - compilable into binary/assembly/your favourite language;
- Felleisen 1990: "Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal".
- But:
  - no programmer writes real programs in assembly language or Turing machines;
  - compilation collapses all specific properties of programming languages, which every programmer feels in his practice.
- Thus, *programs must not be examined up to compilation*, but as source codes, before their compilation.

The compilation fallacy

- The compilation fallacy: all programming languages are "equivalent", since most of them are
  - Turing-complete
  - compilable into binary/assembly/your favourite language;
- Felleisen 1990: "Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal".
- But:
  - no programmer writes real programs in assembly language or Turing machines;
  - compilation collapses all specific properties of programming languages, which every programmer feels in his practice.
- Thus, *programs must not be examined up to compilation*, but as source codes, before their compilation.

The compilation fallacy

- The compilation fallacy: all programming languages are "equivalent", since most of them are
  - Turing-complete
  - compilable into binary/assembly/your favourite language;
- Felleisen 1990: "Comparing the set of computable functions that a language can represent is useless because the languages in question are usually universal".
- But:
  - no programmer writes real programs in assembly language or Turing machines;
  - compilation collapses all specific properties of programming languages, which every programmer feels in his practice.
- Thus, *programs must not be examined up to compilation*, but as source codes, before their compilation.

Theoretical programming languages

What common-sense properties can we (maybe naively) expect from theoretical programming languages?

No syntactic irregularity

- (1) The language should contain no syntactic irregularity: every syntactic rule must be general, and followed in all cases.

No expletiveness

- (2) The language should contain no expletiveness: every sign of the language must have a meaning and be useful (otherwise, there is no use declaring it).

No redundancy

- (3) The language should have no redundancy: there must be only one sign for each fundamental concept.
- Schönfinkel, "On the building blocks of mathematical logic", 1924:

> *It is in the spirit of the axiomatic method [...] that we not only strive to keep the axioms as few and their content as limited as possible but also attempt to make the number of fundamental undefined notions as small as we can; we do this by seeking out thoses notions from which we shall best be able to construct all other notions of the branch of science in question.*

No redundancy

- (3) The language should have no redundancy: there must be only one sign for each fundamental concept.
- Schönfinkel, "On the building blocks of mathematical logic", 1924:

> *It is in the spirit of the axiomatic method [...] that we not only strive to keep the axioms as few and their content as limited as possible but also attempt to make the number of fundamental undefined notions as small as we can; we do this by seeking out thoses notions from which we shall best be able to construct all other notions of the branch of science in question.*

No historical residues

- (4) Theoretical programming languages have no passive historical residues. Every language can be considered as starting from scratch.
- Example:
  - Church uses the symbols Σ and Π for quantification;
  - Martin-Löf 1980 can harmessly reuse these signs with new rules. There is no conflict, since it is a new system, declared and understood as such.

No historical residues

- (4) Theoretical programming languages have no passive historical residues. Every language can be considered as starting from scratch.
- Example:
  - Church uses the symbols Σ and Π for quantification;
  - Martin-Löf 1980 can harmessly reuse these signs with new rules. There is no conflict, since it is a new system, declared and understood as such.

Learning by concepts

- (5) The language is learnt through its abstract definition;
- Example: Gilles Dowek and Jean-Jacques Lévy 2006 for PCF:

```
t = x
  | fun x -> t
  | t t
  | n
  | t + t | t - t | t * t | t / t
  | ifz t then t else t
  | fix x t
  | let x = t in t
```

Learning by concepts

- (5) The language is learnt through its abstract definition;
- Example: Gilles Dowek and Jean-Jacques Lévy 2006 for PCF:

```
t = x
  | fun x -> t
  | t t
  | n
  | t + t | t - t | t * t | t / t
  | ifz t then t else t
  | fix x t
  | let x = t in t
```

Learning by concepts

- Teaching order:
  1. introductory remarks;
  2. BNF syntax;
  3. theorems.

Learning by concepts

- Teaching order:
  1. introductory remarks;
  2. BNF syntax;
  3. theorems.

Learning by concepts

- Teaching order:
  1. introductory remarks;
  2. BNF syntax;
  3. theorems.

What common-sense properties can we expect from theoretical programming languages?

- No syntactic irregularity;
- No expletiveness;
- No redundancy;
- No historical residues;
- Abstract definition.

Real-life programming languages

What properties can we now observe in real-life programming languages?

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Syntactic irregularity

- (1) There are surprises, exceptions in the language.
- Examples in C:
  - `int fact (int n) { ... }`
  - `char foo (int n) { ... }`
  - `void helloworld (void) { ... }`
  - `helloworld () { ... }`
- Such irregularities should not happen in theoretical programming languages, bacause they break generality.

Expletiveness

- (2) Some parts of the program are useless:
  - comments;
  - empty lines and spaces;
- they are ignored by the compiler;
- but they are highly recommended by programmers!

Expletiveness

- (2) Some parts of the program are useless:
  - comments;
  - empty lines and spaces;
- they are ignored by the compiler;
- but they are highly recommended by programmers!

Expletiveness

- (2) Some parts of the program are useless:
  - comments;
  - empty lines and spaces;
- they are ignored by the compiler;
- but they are highly recommended by programmers!

Expletiveness

- (2) Some parts of the program are useless:
  - comments;
  - empty lines and spaces;
- they are ignored by the compiler;
- but they are highly recommended by programmers!

Expletiveness

- Example in UNIX v6:

```
0100 /* fundamental constants: cannot be changed */
0101
0102
0103 #define USIZE   16
0104 #define NULL    0
0105 #define NODEV   (-1)
0106 #define ROOTINO 1
0107 #define DIRSIZ  14
0108
0109
0110 /* signals: dont change */
0111 #define NSIG    20
...
```

Redundancies

- (3) Real-life programming languages, such as C, have a lot of redundancies (syntactic sugar):

```
i = 0;
label:
if (i < 10) {
    commands;
    i = i + 1;
    goto label:
}
```

```
i = 0;
while (i < 10) {
    commands;
    i = i + 1;
}
```

```
for (i = 0; i < 10; i = i+1) {
    commands;
}
```

C

- And also the ternary operator `?:`, `until` (in Perl) ...
- In theoretical languages, this would be a loss of conceptual purity.

Redundancies

- (3) Real-life programming languages, such as C, have a lot of redundancies (syntactic sugar):
-

```
i = 0;
label:
if (i < 10) {
    commands;
    i = i + 1;
    goto label:
}
```

```
i = 0;
while (i < 10) {
    commands;
    i = i + 1;
}
```

```
for (i = 0; i < 10; i = i+1) {
    commands;
}
```

C

- And also the ternary operator `?:`, `until` (in Perl) ...
- In theoretical languages, this would be a loss of conceptual purity.

Redundancies

- (3) Real-life programming languages, such as C, have a lot of redundancies (syntactic sugar):
-

```
i = 0;
label:
if (i < 10) {
    commands;
    i = i + 1;
    goto label:
}
```

```
i = 0;
while (i < 10) {
    commands;
    i = i + 1;
}
```

```
for (i = 0; i < 10; i = i+1) {
    commands;
}
```

C

- And also the ternary operator `?:`, `until` (in Perl) ...
- In theoretical languages, this would be a loss of conceptual purity.

Redundancies

- (3) Real-life programming languages, such as C, have a lot of redundancies (syntactic sugar):
-

```
i = 0;
label:
if (i < 10) {
    commands;
    i = i + 1;
    goto label:
}
```

```
i = 0;
while (i < 10) {
    commands;
    i = i + 1;
}
```

```
for (i = 0; i < 10; i = i+1) {
    commands;
}
```

C

- And also the ternary operator `?:`, `until` (in Perl) ...
- In theoretical languages, this would be a loss of conceptual purity.

Historical residues

- (4) Historicity is present in the languages themselves.
  - retro-compatibility: an older program must remain compilable in modern compilers;
  - some obsolete keywords are arbitrarily forbidden or ignored, just for historical reasons: `register` in C.
- You cannot learn a programming language without by the way learning elements of their history.
- By contrast, theoretical languages should not be weighed down by historical contingences.

Historical residues

- (4) Historicity is present in the languages themselves.
  - retro-compatibility: an older program must remain compilable in modern compilers;
    - some obsolete keywords are arbitrarily forbidden or ignored, just for historical reasons: `register` in C.
- You cannot learn a programming language without by the way learning elements of their history.
- By contrast, theoretical languages should not be weighed down by historical contingences.

Historical residues

- (4) Historicity is present in the languages themselves.
  - retro-compatibility: an older program must remain compilable in modern compilers;
  - some obsolete keywords are arbitrarily forbidden or ignored, just for historical reasons: `register` in C.
- You cannot learn a programming language without by the way learning elements of their history.
- By contrast, theoretical languages should not be weighed down by historical contingences.

Historical residues

- (4) Historicity is present in the languages themselves.
  - retro-compatibility: an older program must remain compilable in modern compilers;
  - some obsolete keywords are arbitrarily forbidden or ignored, just for historical reasons: `register` in C.
- You cannot learn a programming language without by the way learning elements of their history.
- By contrast, theoretical languages should not be weighed down by historical contingences.

Historical residues

- (4) Historicity is present in the languages themselves.
  - retro-compatibility: an older program must remain compilable in modern compilers;
  - some obsolete keywords are arbitrarily forbidden or ignored, just for historical reasons: `register` in C.
- You cannot learn a programming language without by the way learning elements of their history.
- By contrast, theoretical languages should not be weighed down by historical contingences.

Learning by practice

- (5) Languages are not taught with their BNF syntax, but with *hello world*s.
- Example in Kernighan and Ritchie 1978:

```
#include <stdio.h>
main()
{
  printf("Hello world!\n");
}
```

- Example in L. Wall 1991:

```
$phrase = "Howdy, world!\n";    # Set a variable
print $phrase;                  # Print the variable
```

Learning by practice

- (5) Languages are not taught with their BNF syntax, but with *hello world*s.
- Example in Kernighan and Ritchie 1978:

```
#include <stdio.h>
main()
{
  printf("Hello world!\n");
}
```

- Example in L. Wall 1991:

```
$phrase = "Howdy, world!\n";    # Set a variable
print $phrase;                  # Print the variable
```

Learning by practice

- (5) Languages are not taught with their BNF syntax, but with *hello world*s.
- Example in Kernighan and Ritchie 1978:

```
#include <stdio.h>
main()
{
  printf("Hello world!\n");
}
```

- Example in L. Wall 1991:

```
$phrase = "Howdy, world!\n";     # Set a variable
print $phrase;                   # Print the variable
```

Learning by practice

- Teaching order:
  1. *Hello world* code;
  2. compilation and observation;
  3. explanation of the terms (`include`, `main()`, `printf`...).
  4. The (facultative) BNF syntax is given long after the first definition of the language — only in appendix, — and only for compiler programmers (nobody reads it).

Learning by practice

- Teaching order:
  1. *Hello world* code;
  2. compilation and observation;
  3. explanation of the terms (`include`, `main()`, `printf`...).
  4. The (facultative) BNF syntax is given long after the first definition of the language — only in appendix, — and only for compiler programmers (nobody reads it).

Learning by practice

- Teaching order:
  1. *Hello world* code;
  2. compilation and observation;
  3. explanation of the terms (`include`, `main()`, `printf`...).
  4. The (facultative) BNF syntax is given long after the first definition of the language — only in appendix, — and only for compiler programmers (nobody reads it).

Learning by practice

- Teaching order:
  1. *Hello world* code;
  2. compilation and observation;
  3. explanation of the terms (`include`, `main()`, `printf`...).
  4. The (facultative) BNF syntax is given long after the first definition of the language — only in appendix, — and only for compiler programmers (nobody reads it).

Real-life programming languages and natural languages

- Real-life programming languages possess many properties which would not be expected in a theoretical programming language:
  1. syntactic irregularity;
  2. expletiveness;
  3. redundancies;
  4. historical residues (Ancient Chinese in modern Chinese, Latin in Italian and French);
  5. learning by practice.
- They share all these properties with *natural languages*. And in natural languages, these are even positive properties: they give birth to style.

Real-life programming languages and natural languages

- Real-life programming languages possess many properties which would not be expected in a theoretical programming language:
  1. syntactic irregularity;
  2. expletiveness;
  3. redundancies;
  4. historical residues (Ancient Chinese in modern Chinese, Latin in Italian and French);
  5. learning by practice.
- They share all these properties with *natural languages*. And in natural languages, these are even positive properties: they give birth to style.

Consequences

Consequences:

- One could try to transpose to the study of programming languages some methods initially devoted to natural languages, providing a "philology" of programming languages;
- this leads us to reverse Chomsky's thesis about formal and natural languages.

Consequences

Consequences:

- One could try to transpose to the study of programming languages some methods initially devoted to natural languages, providing a "philology" of programming languages;
- this leads us to reverse Chomsky's thesis about formal and natural languages.

Chomsky: English as a formal language

- Chomsky, *Syntactic Structures* (1957):
  - Σ : # *Sentence* #
  - F:

$$
\begin{array}{lcl}
\mathit{Sentence} & \rightarrow & \mathit{NP} + \mathit{VP} \\
\mathit{VP} & \rightarrow & \mathit{Verb} + \mathit{NP} \\
\mathit{NP} & \rightarrow & \left\{ \begin{array}{l} \mathit{NP}_{sing} \\ \mathit{NP}_{pl} \end{array} \right\} \\
\mathit{NP}_{sing} & \rightarrow & T + N + \varnothing \\
\mathit{NB}_{pl} & \rightarrow & T + N + S \\
T & \rightarrow & \mathit{the} \\
N & \rightarrow & \mathit{man}, \mathit{ball}, \mathit{etc}. \\
\mathit{Verb} & \rightarrow & \mathit{Aux} + V \\
V & \rightarrow & \mathit{hit}, \mathit{take}, \mathit{walk}, \mathit{read}, \mathit{etc}. \\
\mathit{Aux} & \rightarrow & C(M)(\mathit{have} + \mathit{en})(\mathit{be} + \mathit{ing}) \\
M & \rightarrow & \mathit{will}, \mathit{can}, \mathit{may}, \mathit{shall}, \mathit{must}
\end{array}
$$

A naturalization of formal languages

- Chomsky suggests a formalization of natural languages.
- Instead, we are suggesting a naturalization of at least some formal languages.
- But this attempt must be distinguished from other works about:
  - expressiveness (Felleisen 1990, Mitchell 1991);
  - levels of abstraction.

A naturalization of formal languages

- Chomsky suggests a formalization of natural languages.
- Instead, we are suggesting a naturalization of at least some formal languages.
- But this attempt must be distinguished from other works about:
  - expressiveness (Felleisen 1990, Mitchell 1991);
  - levels of abstraction.

A naturalization of formal languages

- Chomsky suggests a formalization of natural languages.
- Instead, we are suggesting a naturalization of at least some formal languages.
- But this attempt must be distinguished from other works about:
  - expressiveness (Felleisen 1990, Mitchell 1991);
  - levels of abstraction.

Formal theories of expressiveness

- Felleisen 1990, Mitchell 1991: formal criteria to define and compare the expressiveness of programming languages.
- Basically, they define expressiveness using morphisms between languages (language translation):

> *Given two universal programming languages that only differ by a set of programming constructs, $\{c_1, ..., c_n\}$, the relation holds if the additional constructs make the larger language more expressive than the smaller one. Here "more expressive" means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program. (Felleisen)*

Formal theories of expressiveness

- Felleisen 1990, Mitchell 1991: formal criteria to define and compare the expressiveness of programming languages.
- Basically, they define expressiveness using morphisms between languages (language translation):

> *Given two universal programming languages that only differ by a set of programming constructs, $\{c_1, ..., c_n\}$, the relation holds if the additional constructs make the larger language more expressive than the smaller one. Here "more expressive" means that the translation of a program with occurrences of one of the constructs c_i to the smaller language requires a global reorganization of the entire program. (Felleisen)*

Formal theories of expressiveness

- But a part of the expressiveness (e.g. expletiveness) can not be described with *formal* tools such as morphisms between languages.

Levels of abstraction

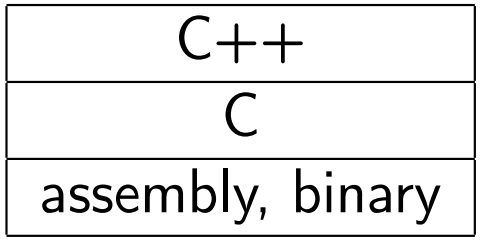
- Levels of abstraction: classification of programming languages based on their abstraction.

C++
C
assembly, binary

- But there are other criteria.

Levels of abstraction

- Levels of abstraction: classification of programming languages based on their abstraction.



- But there are other criteria.

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- How does a programmer choose the language for a given task?

1. type of programming language: logic programming, query language, web programming...
2. vertical criteria (abstraction):
   1. economy: proximity with the machine (for example, assembly for some parts of operating systems, such as drivers);
   2. expressiveness: proximity with human representations;
3. but there are also quite shameful horizontal criteria:
   1. disciplinary traditions: FORTRAN in physics;
   2. matters of taste: some people prefer indentation, others prefer brackets (Perl/Python).

```
def fact(x):
    if x < 2:
        return 1
    else:
        return x * fact(x-1)
```

- Neither expressiveness not levels of abstraction explain the whole variety of programming languages. Many of their aspects escape technical criteria, and involve more general linguistics.
- Two main aspects will be described:
  1. historical aspects;
  2. stylistic aspects.

Historical aspects

Historical aspects:

- etymology:
  - terms used in programming language do not always have a meaning in themselves, but can allude to former languages:
    - the C-like syntax of C++, Java...
    - even the `lambda` word for anonymous functions in Lisp and Python!
- importation phenomena:
  - τὸ in Latin (Spinoza: "τὸ *ens*"), "par excellence" in English;
  - object-oriented programming in Common Lisp and PHP, C++ as a C with classes...
- living languages: Perl 1.0 (1987), Perl 2.0 (1988), Perl 3 (1989), Perl 5 (1994), Perl 6 (since 2000)...
- dead or zombie languages (when they have no speaker anymore, and stopped their evolution): B, Smalltalk...

Historical aspects

Historical aspects:

- etymology:
  - terms used in programming language do not always have a meaning in themselves, but can allude to former languages:
    - the C-like syntax of C++, Java...
    - even the `lambda` word for anonymous functions in Lisp and Python!
- importation phenomena:
  - τὸ in Latin (Spinoza: "τὸ *ens*"), "par excellence" in English;
  - object-oriented programming in Common Lisp and PHP, C++ as a C with classes...
- living languages: Perl 1.0 (1987), Perl 2.0 (1988), Perl 3 (1989), Perl 5 (1994), Perl 6 (since 2000)...
- dead or zombie languages (when they have no speaker anymore, and stopped their evolution): B, Smalltalk...

Historical aspects

Historical aspects:

- etymology:
  - terms used in programming language do not always have a meaning in themselves, but can allude to former languages:
    - the C-like syntax of C++, Java...
    - even the `lambda` word for anonymous functions in Lisp and Python!
- importation phenomena:
  - τὸ in Latin (Spinoza: "τὸ *ens*"), "par excellence" in English;
  - object-oriented programming in Common Lisp and PHP, C++ as a C with classes...
- living languages: Perl 1.0 (1987), Perl 2.0 (1988), Perl 3 (1989), Perl 5 (1994), Perl 6 (since 2000)...
- dead or zombie languages (when they have no speaker anymore, and stopped their evolution): B, Smalltalk...

Historical aspects

Historical aspects:

- etymology:
  - terms used in programming language do not always have a meaning in themselves, but can allude to former languages:
    - the C-like syntax of C++, Java...
    - even the `lambda` word for anonymous functions in Lisp and Python!
- importation phenomena:
  - τὸ in Latin (Spinoza: "τὸ *ens*"), "par excellence" in English;
  - object-oriented programming in Common Lisp and PHP, C++ as a C with classes...
- living languages: Perl 1.0 (1987), Perl 2.0 (1988), Perl 3 (1989), Perl 5 (1994), Perl 6 (since 2000)...
- dead or zombie languages (when they have no speaker anymore, and stopped their evolution): B, Smalltalk...

Historical aspects

Historical aspects:

- etymology:
  - terms used in programming language do not always have a meaning in themselves, but can allude to former languages:
    - the C-like syntax of C++, Java...
    - even the `lambda` word for anonymous functions in Lisp and Python!
- importation phenomena:
  - τὸ in Latin (Spinoza: "τὸ *ens*"), "par excellence" in English;
  - object-oriented programming in Common Lisp and PHP, C++ as a C with classes...
- living languages: Perl 1.0 (1987), Perl 2.0 (1988), Perl 3 (1989), Perl 5 (1994), Perl 6 (since 2000)...
- dead or zombie languages (when they have no speaker anymore, and stopped their evolution): B, Smalltalk...

Stylistics aspects

Stylistics aspects:

- normative grammar:
  - avoid GOTOs and spaghetti code;
  - use comments, empty lines and spaces;
  - compiler Warnings (`gcc -Wall -pedantic`)
  - elegance of algorithms (whatever be their complexity);
  - books of recommendations: *Beautiful Code* (A. Oram and G. Wilson), *Programming Pearls* (J. L. Bentley)...
- the natural ontologies of programming languages:
  - ontology of pure transformations in functional programming;
  - animism of object-oriented programming;
  - ...

Stylistics aspects

Stylistics aspects:

- normative grammar:
  - avoid GOTOs and spaghetti code;
  - use comments, empty lines and spaces;
  - compiler Warnings (`gcc -Wall -pedantic`)
  - elegance of algorithms (whatever be their complexity);
  - books of recommendations: *Beautiful Code* (A. Oram and G. Wilson), *Programming Pearls* (J. L. Bentley)...
- the natural ontologies of programming languages:
  - ontology of pure transformations in functional programming;
  - animism of object-oriented programming;
  - ...

Stylistics aspects

Stylistics aspects:

- normative grammar:
  - avoid GOTOs and spaghetti code;
  - use comments, empty lines and spaces;
  - compiler Warnings (`gcc -Wall -pedantic`)
  - elegance of algorithms (whatever be their complexity);
  - books of recommendations: *Beautiful Code* (A. Oram and G. Wilson), *Programming Pearls* (J. L. Bentley)...
- the natural ontologies of programming languages:
  - ontology of pure transformations in functional programming;
  - animism of object-oriented programming;
  - ...

Stylistic aspects

- ... and even literature!
- L. Spitzer 1948:

  > *Now, since the best document of the soul of a nation is its literature, and since the latter is nothing but its language as this is written down by elect speakers, can we perhaps not hope to grasp the spirit of a nation in the language of its outstanding works of literature?*

- What would be epic poetry for programming languages?
- Operating systems! Big source codes which define all the ontology of the machine with which the user will interact: notions of process, file, kernel...

Stylistic aspects

- ... and even literature!
- L. Spitzer 1948:

  > *Now, since the best document of the soul of a nation is its literature, and since the latter is nothing but its language as this is written down by elect speakers, can we perhaps not hope to grasp the spirit of a nation in the language of its outstanding works of literature?*

- What would be epic poetry for programming languages?
- Operating systems! Big source codes which define all the ontology of the machine with which the user will interact: notions of process, file, kernel...

Stylistic aspects

- ... and even literature!
- L. Spitzer 1948:

  > *Now, since the best document of the soul of a nation is its literature, and since the latter is nothing but its language as this is written down by elect speakers, can we perhaps not hope to grasp the spirit of a nation in the language of its outstanding works of literature?*

- What would be epic poetry for programming languages?
- Operating systems! Big source codes which define all the ontology of the machine with which the user will interact: notions of process, file, kernel...

- Philosophers of mathematics are not afraid of reading mathematical texts.
- Philosophers of computer science should not be afraid of reading source codes.

- Philosophers of mathematics are not afraid of reading mathematical texts.
- Philosophers of computer science should not be afraid of reading source codes.

Conclusion

- Are real-life programming languages failed theoretical languages? = Are horses failed unicorns?
- Theoretical programming languages deliberately miss some features of real-life programming languages.
- Even formal tools such as expressiveness do not capture all of them.
- By contrast, some linguistic concepts (etymology, normative grammar, importation, literature...) can help describing them.

Conclusion

- Are real-life programming languages failed theoretical languages? = Are horses failed unicorns?
- Theoretical programming languages deliberately miss some features of real-life programming languages.
- Even formal tools such as expressiveness do not capture all of them.
- By contrast, some linguistic concepts (etymology, normative grammar, importation, literature...) can help describing them.

Conclusion

- Are real-life programming languages failed theoretical languages? = Are horses failed unicorns?
- Theoretical programming languages deliberately miss some features of real-life programming languages.
- Even formal tools such as expressiveness do not capture all of them.
- By contrast, some linguistic concepts (etymology, normative grammar, importation, literature...) can help describing them.

Conclusion

- Are real-life programming languages failed theoretical languages? = Are horses failed unicorns?
- Theoretical programming languages deliberately miss some features of real-life programming languages.
- Even formal tools such as expressiveness do not capture all of them.
- By contrast, some linguistic concepts (etymology, normative grammar, importation, literature...) can help describing them.

Conclusion

- Just like recent philosophy of mathematics, the philosophy of programming languages can describe the gap which sometimes appears between foundations and practices.
- This can even retrospectively show us some hidden properties of theoretical programming languages, for there is a practice of foundations.

Conclusion

- Just like recent philosophy of mathematics, the philosophy of programming languages can describe the gap which sometimes appears between foundations and practices.
- This can even retrospectively show us some hidden properties of theoretical programming languages, for there is a practice of foundations.

Bibliography

- N. Chomsky 1957, *Syntactic Structures*.
- G. Dowek and J.-J. Lévy 2006, *Introduction à la théorie des langages de programmation*.
- M. Felleisen 1990, "On the Expressive Power of PLs".
- Kernighan & Ritchie 1978, *The C Programming Language*.
- J. Mitchell 1991, "On Abstraction and the Expressive Power of PLs".
- P. Rechenberg 1990, "PLs as Thought Models".
- L. Spitzer 1948, "Linguistics and Literary History".
- B. Stroustrup 2000, *The C++ Programming Language*.
- L. Wall et al. 1991, *Programming Perl*.
- G. White 2008, "The Philosophy of Computer Languages".