Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Concepts of Programming Languages

Introduction

Dr. Amin Allam

[For more details, refer to "Concepts of Programming Languages" by *Robert Sebesta*]

1 Benefits of studying concepts of programming languages

• *Increasing the ability to describe programs:* Programmers are usually not aware of all features supported by the programming languages they use. By learning the capabilities of programming languages, programmers can use such capabilities to simplify thinking, designing, and writing programs. Programmers may also simulate such capabilities within a specific programming language which does not support them. As a result, the process of software development is simplified.

• *Increasing the ability to choose appropriate programming language:* By understanding the basic concepts and features of various programming languages, a programmer will have the ability to choose the most appropriate programming language for a considered project, without detailed previous knowledge of that programming language.

• *Increasing the ability to learn new language:* Understanding the fundamental concepts and vocabulary of programming languages simplifies significantly the process of learning new programming languages and reading programming language descriptions and literature.

• *Understanding the significance of implementation:* By understanding how programming concepts are implemented, a programmer can utilize the programming language features more efficiently, or at least to know the program parts which can possibly cause inefficiency. Also, using such knowledge, a programmer can find and fix certain kind of program bugs (errors).

• *Improving the design of programming languages:* By understanding various programming language features, programmers can understand the benefits and limitations of existing programming languages. As a result, they may come up with a better design of new programming languages, or improving the design of existing ones.

2 Programming domains

• *Scientific applications:* Engineering applications with large number of arithmetic computations.
• *Business applications:* Facilitates producing reports using decimal arithmetic and character data.
• *Artificial intelligence:* Symbolic computations suitable for deducing information from data.
• *Systems programming:* Operating systems control computer devices and external devices.
• *Web software:* Markup languages and scripting languages for dynamic web contents.

3 Evaluating programming languages

In order to be able to recognize the advantages and limitations of programming languages, or the suitability of a programming language to a considered application or project, we should be able to evaluate programming languages. For this purpose, we consider the following *evaluation criteria*:

- *Readability:* The ease with which programs of a language can be read and understood.
- *Writability:* How easily a language can be used to create programs for a chosen problem domain.
- *Reliability:* The ability of programs to conform with its specifications under all conditions.
- *Maintainability:* The ease of correcting, modifying, and enhancing existing programs.
- *Efficiency:* The amount of memory and processor time consumed when running programs.
- *Portability:* The ability to run programs on different operating systems and hardware.
- *Generality:* The applicability of a language to a wide range of applications.

Some of the above evaluation criteria are strongly correlated. For example, *readability* usually (but not necessarily) enhances *writability* because writing a program can be viewed as modifying a small program several times until it is finished. Thus, if it can be read easily, it can be easily written as well. If a program is *readable* and *writable*, the ratio of errors in the program will be small, and then its *reliability* and *maintainability* will improve.

The following language characteristics contribute to some of the above evaluation criteria:

3.1 Simplicity, expressivity, and orthogonality

A programming language is *simple* if it consists of a small number of basic constructs to be learned. *Simplicity* allows programmers to easily learn the programming language. Therefore, it enhances *readability* because the program reader needs to save in his mind a small amount of information to be able to understand the program. A programming language is *expressive* if it consists of a set of constructs that allow compact programs to perform a lot of computations. Therefore, it enhances *writability*. Consider the following C++ statements:

```
a=a+1;
a+=1;
a++;
++a;
```

The above 4 statements are different syntactic alternatives to do the same thing, which is adding 1 to the variable a. Therefore, for this operation, C++ is *expressive* but not *simple*. This requires a programmer to learn the 4 ways to be able to read C++ programs, so it reduces *readability*. A programmer can write a compact statement such as `a++;` which enhances *writability*.

Operator overloading is another feature which improves *expressivity* since a programmer can overload an existing operator to do a new action. However, it reduces *simplicity* since it adds a new functionality to the language which should be learned by whoever needs to read such program. Note that if the programmer carefully uses *operator overloading* such that it behaves similarly to the original behaviour of the programming language but with new data types, that will not reduce *simplicity* but still enhances *expressivity*. This is called *orthogonality*.

Orthogonality means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to define the language behaviour, such that every legal combination of primitives is meaningful. So, *orthogonality* combines *simplicity* and *expressivity*. For example, the C++ statement `a+b` can be used to add two integers, two floating point numbers, or an integer and a floating point number, depending on the data types of the variables `a` and `b`. The language construct is considered *simple* because the programmer does not need to save in his mind anything except that the + operator adds two things and returns the result. However, the construct is *expressive* because it can be used to add several different pairs of things. Contrary, in some assembly languages, different addition instructions are used according to the parameter types. Another example of *lack* of *orthogonality* in C++ is that all parameters can be passed by value, except for arrays which can never be passed by value. A programmer needs to save in his mind that array elements are actually passed by reference although they look as if they are passed by value.

3.2 Control statements

The existence of adequate control statements in a programming language significantly enhances its *readability* and *writability*. For example, consider the following C++ code:

```
inc=100;
while(inc>3)
{
   while(sum<=1000)
   {
      sum+=inc*inc;
   }
   inc/=2;
}
```

If the language does not contain a `while` statement, the code will probably look like:

```
inc=100;
loop1:
   if(inc<=3) goto out;
loop2:
   if(sum>1000) goto next;
   sum+=inc*inc;
   goto loop2;
next:
   inc/=2;
   goto loop1;
out:
```

The presence of control statements does not guarantee that programs will be readable. The programmer should use them intelligently and meaningfully, and should provide adequate padding and spacing within the control statements.

3.3 Data types and structures

The presence of good ways to define data types and structures in a language improves *readability*. For example, if the language contains a boolean data type, an indicator flag can be set by:

```
timeOut=true;
```

Otherwise, we will use a numeric value whose meaning is not very clear:

```
timeOut=1;
```

Also, to construct an array of 20 students in C++:

```
struct Student
{
   char name[100];
   int level;
   double gpa;
};
Student s[20];
```

If it is not possible to define new data types in C++, we will be obligated to use arrays that do not seem to have any connection, which reduce *readability*.

```
char name[20][100];
int level[20];
double gpa[20];
```

3.4 Syntax design

The syntax or form of language elements has a significant effect on *readability*. In some old languages there was a restriction on *identifier* length, which reduces *readability*. The existence of specific *special words* may enhance *readability*, such as using `end loop` and `end if` instead of } to close `while` and `if` blocks. If a language allows using *special words* as variable names, *readability* may be reduced.

The appearance of statements and special words should help understand its purpose to improve *readability*. This principle is somehow violated if two language constructs are identical or similar in appearance but have different meaning. For example, in C++, the meaning of the reserved word `static` depends on its context. If used to define variable inside a function, it means that the variable is created before runtime and remains allocated until the end of the program. If used to define a variable outside all functions, it means that the variable is visible only within the file where the definition appears. Another example in C++, the reserved word `const` has the meaning of *'constant'* if the variable is initialized with a constant value, otherwise it has the meaning of *'read only'*. The mentioned examples reduce *readability*.

Also, a language supporting high degree of *abstraction* has high *writability*. *Abstraction* is the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. It will be discussed in an upcoming lecture later on.

3.5 Type checking

Type checking is testing for type errors in a given program, either by the compiler or during program execution. A language which checks for type errors is much more *reliable* than a language which does not. A language which checks for type errors in compile time is much more *reliable* than a language which checks for them during program execution, because if the compiler detects a type error, the programmer will fix it before providing the user with the program.

For example, suppose that a program passes a floating point variable as a parameter to a function which expects an integer variable. If the compiler did not detect such type error to do implicit conversion or at least to signal an error to the programmer, the function will not be able to correctly detect the intended passed value.

3.6 Exception handling

Exception handling is the ability of a program to intercept runtime errors, take corrective actions, and then continue according to the program specifications. A language supporting *exception handling* is *reliable* if the programmer effectively utilizes this feature.

3.7 Aliasing

Aliasing is having two or more distinct names in a program that can be used to access the same memory cell. *Aliasing* is an important feature which is required in several situations in programming, such as parameter passing and shared objects. However, the excessive usage of *aliasing* may reduce *reliability* since it causes program bugs if the programmer forgot that changing the value of a variable affects the value of its *alias* as well.

4 Implementation methods

To run a program, one of the following methods is used, according to the programming language:

4.1 Compilation

Programs are *translated* into machine language only once, by another program dedicated for this purpose called a *compiler*. A *compiler* takes a program written in the *source language* of a specific programming language, and translates it into a program written in machine language (the executable) suitable to the used machine and the running operating system. Once the program is compiled, it can be run several times on that specific machine and operating system configuration without any additional compilation. The compiler usually performs important optimizations to increase the runtime efficiency of the program. Also, since the compiler has the chance to detect several errors during compilation time, program reliability is expected to improve.

Compilation is preferred for desktop applications and embedded systems, because such programs run several times, therefore runtime efficiency and reliability are needed.

4.2 Interpretation

Programs are *interpreted* into machine language line-by-line by another program dedicated for this purpose called an *interpreter*. To execute a program, an *interpreter* takes a program written in the *source language* of a specific programming language, and executes its statements line-by-line. That is, running the program is mixed with its interpretation. The *interpreter* takes the first line of the program, interprets it to machine language, then executes it, then determines the next line according to the execution scenario, then interprets the next line, and so on. This process is done each time the program runs. Since running is mixed with interpretation, interpreted programs are much slower than compiled programs. For example, each statement within the body of a loop is interpreted to machine language from scratch every time it is executed. Some languages, such as *scripting languages*, can be interpreted but can never be compiled.

Interpretation is preferred for web client applications, because such programs are created on-the-fly according to a response from the web server applications. Since web client applications run very few times when that particular web page is browsed, and the nature of such programs is simple, compilation will not significantly improve runtime efficiency or reliability.

5 Programming languages categories

The following programming language categories are not disjoint. One language can be classified into several categories. For example, *Javascript* is imperative, object-oriented, and scripting.

• *Imperative/Procedural languages:* Ordinary languages which consist mainly of data and procedures (sequences of statements to be executed), and are considered as high level simplification of machine languages for von-Neumann machines. *Examples: Assembly*, *Fortran*, *C*.

• *Object-oriented languages:* Subset of imperative languages which encapsulate processing with data objects and control access to data, and add inheritance and dynamic method binding. *Examples: Smalltalk*, *C++*, *Java*.

• *Scripting languages:* Subset of imperative languages which are interpreted, not compiled. *Examples: Javascript*, *Perl*, *Python*.

• *Declarative/Functional/Logic languages:* Rule-based languages where rules are specified in no particular order, and no particular procedure is specified. The language execution system must choose an algorithm and an execution order to produce the required results. *Examples: Prolog*, *LISP*, *Haskell*.

• *Markup languages:* Languages used to describe something. They are similar to declarative languages in avoiding procedures, with the difference that rules are replaced by descriptive items. Some of these languages include few procedures. *Examples: XHTML* (specifies the layout of information in web documents), *XML*, *Latex*.

• *Special-purpose languages:* Languages dedicated for specific applications. They have narrow applicability but they are very effective for their purpose. *Mathematical programs should be written in mathematical notation. Data processing programs should be written in English statements* (Wexelblat). *Examples: COBOL* (computing business records), *GPSS* (systems simulation).