Symbolic programming using Common Lisp

Robert Strandh

compiled by Loic Lacomme, October 25, 2000

Contents

1. Introduction

In this document, we describe symbolic programming techniques using the Common Lisp language.

There are a number of chapters. We recommend you read them in this order:

1. Why Common Lisp. The reason use Common Lisp.
2. Interpreted languages. Two plausible definitions of "interpreted language" and why Common Lisp is not more interpreted than other languages.
3. Difference with respect to Scheme. For those who know the Scheme language, a list of the major differences between Scheme and Common Lisp.
4. The psychology of learning. An explanation to an observed phenomenon that many people do not want to learn new things.
5. Programming language syntax. Common Lisp has a two-level syntax. We explain what that means, and why it is crucial.
6. Autonomous functions. We define what an autonomous function is, and why it is good to write autonomous functions.
7. Side-effect free functions. We define what a side-effect free function is, and why it is good to write side-effect free functions.
8. Indentation of Common Lisp. Programmers of Common Lisp have an agreement on how to format and indent code. Here is a summary of the rules.
9. Top-down programming. Why top-down programming is usually not what you want.
10. Bottom-up programming. Why bottom-up programming is usually what you want.
11. Building specialized languages. A programming technique that is well adapted to program maintenance.
12. Object-oriented programming. Fundamental terminology of object-oriented programming and how it applies to Common Lisp and other languages.

There is also a glossary.

2. Why we use Common Lisp

2.1. Introduction

Many students ask the question why we use Common Lisp in teaching programming. Here, we try to answer that question.

There are many reasons for students to question the choice of Common Lisp. We therefore structure this section around different reasons students give for not wanting Common Lisp.

2.2. The industry argument

2.2.1. The argument

Common Lisp is not used in industry (or rather, Common Lisp is never seen in the advertisements for employment). The university should teach programming languages that are actually used, so that students have useful knowledge when they graduate.

2.2.2. The refutation

First, Common Lisp *is* used. Several major commercial software packages are written in Common Lisp or some other Lisp variant. It is hard to know in what language commercial software is written (since the user should not have to care), but there are a few that are well known. Interleaf, a documentation system, is written in Lisp. So is AutoCAD, a system for computer-aided design. Both are major applications in their domains. While not a commercial software system, Emacs is an important system written in Lisp.

But even if Common Lisp were not used at all in industry, this would not be a good argument. The level of sophistication of the software industry is quite low with respect to programming languages, tools, and methods. The university should teach advanced languages, tools and methods with the hope of having industry use them one day, as opposed to teaching bad ones that happen to be used today. Students who want training in particular tools that happen to be demanded at the moment, should quit the university and apply for more specific training programs.

Besides, whether a language such as Common Lisp is used in industry depends a lot more on the individual student than on industry. There is a widespread myth among students that industry is this a monolithic entity whose tools and methods cannot be altered. In reality, industry consists of people. Whatever industry uses is whatever the people working there use. Instead of refusing to learn sophisticated tools and techniques, the student can resolve to try to become one of the forerunners in industry after graduation.

2.3. The readability argument

2.3.1. The argument

Common Lisp notation is unnatural, which makes it hard to understand. Better, then, to use a more natural notation that more people can understand, and that is easier to learn.

2.3.2. The refutation

Depending on your definition of *natural* either all notations are natural, or all notations are unnatural. The correct term would be *familiar*, but then the argument becomes an argument for using only familiar notations. University education is not about requiring the smallest effort possible on the part of the students. Instead it is about teaching tools and methods that allow the student to become more efficient *once knowledge about these tools and methods has been acquired.*

Usually, the learning period for an advanced tool is relatively short compared to the savings in time obtained by long-term use later on. In the case of Common Lisp, the increased efficiency compared to other languages is so great that it easily compensates for any learning period.

There is widespread belief among students that it is intrinsically hard to count parentheses in Lisp. In reality, Lisp users do not count parentheses. A sophisticated text editor such as Emacs assists the programmer with parenthesis matching, and a human reader of Lisp code looks only at the indentation, never at the number of parentheses.

2.4. The efficiency argument

2.4.1. The argument

Common Lisp is an interpreted language, and thus slow. One should use as efficient languages as possible, so Common Lisp is out of the question.

2.4.2. The refutation

First of all, Common Lisp is *not* an interpreted language. In fact, there is not even a reasonable definition of "interpreted language".

Second, most Common Lisp compilers are capable of generating very good, very fast code. Obtaining fast code may require the user to add *declarations* that assist the compiler in optimizing the code, so fast object code is not obtained from naïve source code.

Also, the students have a tendency to exaggerate the importance of speed. There are quite a number of applications that use very slow systems, such as Perl, Shell, Tcl/Tk, Awk, etc. It is simply not true that maximum speed is always required.

2.5. The graphics argument

2.5.1. The argument

Common Lisp does not have advanced graphics functions. Applications with graphic user interfaces are more and more common, so Common Lisp cannot be used for modern applications.

2.5.2. The refutation

Common Lisp has perhaps the most advanced library for writing graphic user interfaces in existence. Its name is CLIM (Common Lisp Interface Manager), and can be bought from major vendors of Lisp technology. Unfortunately, there is not yet a free version of CLIM available (although it is being worked on).

3. What is an interpreted language?

There are only two reasonable definitions of "interpreted language" that I can think of:

1. A language that *can* be implemented with an interpreter,
2. A language that *must* be implemented with an interpreter.

In the first case, all languages are interpreted. In the second case, no language is interpreted.

Sometimes, students confuse *interpreted* and *interactive*, i.e. they think that whenever there is an interactive loop such as the Lisp `read-eval-print` loop, there must also be an interpreter. That is false. The `eval` part can very well be implemented with a compiler.

Sometimes, students think that even though it is *possible* to implement Common Lisp (or Scheme) with a compiler, it is *usually* done with an interpreter, so that *most* implementations are slow. This might be the case for Scheme systems written by students, but is never the case for professional systems. Almost every Common Lisp system in existence uses a compiler.

4. Differences between Scheme and Common Lisp

This section is intended for students who already know Scheme. We describe the most important differences between Scheme and Common Lisp.

4.1. Size of the language

Common Lisp is a large language compared to Scheme, and that is good. Many things in Common Lisp that are missing in Scheme, are features that are frequently needed by many programmers. Having these features standardized is a good thing, as opposed to having to add them, often in an ad-hoc way, for each application.

Some of these features include: packages, arrays, structures, hash tables, object-oriented programming features, a powerful macro feature for extending the language, generalized assignment, and much more.

4.2. Numbers

The Scheme specification allows an implementation of Scheme to have very little support for numbers. In fact, it would be acceptable to have only floating point numbers and still call it Scheme.

Common Lisp, by contracts, requires every implementation to support floating point numbers, rational numbers, integers of arbitrary precision, complex floating point number, and complex rational numbers. This fact makes programs that use numbers more portable.

4.3. Tail call optimization

In Scheme, tail calls must be optimized by the compiler, so that no additional stack space is consumed. Thus, in Scheme, iteration is a special case of tail recursion. You can count on the Scheme compiler (or interpreter) to convert tail calls to simple jumps.

No such rule exists in Common Lisp, although many Common Lisp compilers actually do optimize tail calls. It is therefore less frequent in Common Lisp to use tail recursion for (say) traversing lists. Instead, Common Lisp has a rich set of iteration primitives for exactly that purpose.

4.4. Program/data equivalence

A major problem with Scheme that does not exist in Common Lisp is that the Scheme language does not define an internal representation for programs, although to be fair, all Scheme implementations do.

For instance, in Scheme, there is a difference between:

```
(define (f x y . (z))
  ...)
```

which is syntactically illegal, and:

```
(define (f x y z)
  ...)
```

which syntactically legal. As data (if read by the `read` function) they are the same. Thus, Scheme does not require code to be read by the `read` function.

Common Lisp, on the other hand, does require two such forms to be equivalent. This feature is the very basis of the Common Lisp macro system, which uses Lisp data to represent and manipulate Lisp code.

Since no such equivalence exists in Scheme, one can argue that it is impossible to ever have a good macro system for Scheme. In practice, of course, nearly all implementations do define the equivalence and do contain macro systems.

4.5. Multiple namespaces

The difference that is perhaps the most difficult for students to get used to is the fact that Common Lisp global function names are different from names of other data objects. We say that Common Lisp has *multiple name spaces* whereas Scheme has a single name space.

The main idea in Common Lisp is that the name `fun` in an expression such as:

```
(fun arg)
```

is treated differently from the name `arg`, simply because it is the first subexpression of a function call. Instead of using the normal value of the variable `fun`, its *function value* is used. In normal circumstances, this works the same as Scheme. However, as soon as something slightly more complicated is attempted, the difference is visible.

Perhaps the most obvious difference is that in Common Lisp, it is possible to use the name of a function as an ordinary value without shadowing the function. This is possible in Common Lisp, but not in Scheme:

```
(let ((list '(1 2 3)))
  (list list))
```

In the expression `(list list)`, the first occurrence of `list` refers to the built-in function of that name, and the second refers to the local variable created by `let`.

On the other hand, this is possible in Scheme but not in Common Lisp:

```
(let ((fun (compute-a-function)))
  (fun x y))
```

Here, Common Lisp will search for a global function definition of `fun` (which probably does not exist), whereas in Scheme, the value of the local variable will be used. To do the same thing in Common Lisp, we need to use a built-in function called `funcall` like this:

```
(let ((fun (compute-a-function)))
  (funcall fun x y))
```

Now, the name `fun` is in an argument position, so the value of the local variable is used and passed to `funcall`. The built-in function `funcall` then *applies* its first argument (as a function) to the rest, which is what was intended.

We have seen how Common Lisp through the built-in function `funcall` is able to call a function that is the value of an ordinary variable. We must also solve the inverse problem, i.e., obtain the function object that is not the value of an ordinary variable, but only the global function value of some name such as `car`.

This is possible in Scheme:

```
(map car l)
```

but not in Common Lisp, since typically, the variable `car` would not have an ordinary value. The solution is a special form called `function`, but usually abbreviated `#'`. Here is how to write the same thing in Common Lisp (also taking into account the fact that the Scheme function `map` is called `mapcar` in Common Lisp):

```
(mapcar #'car l)
```

4.6. Minor variations in syntax

There are some minor differences in syntax between Scheme and Common Lisp. To define a procedure in Scheme, you would use the `define` special form like this:

```
(define (f l)
  (g (reverse l)))
```

whereas in Common Lisp, you would use the `defun` special form with slightly different syntax:

```
(defun f (l)
  (g (reverse l)))
```

Other differences exist, like `set!` in Scheme and `setq` in Common Lisp, and `map` in Scheme which does not correspond to the Common Lisp function with the same name, but instead to the Common Lisp function `mapcar`

5. The psychology of learning

I have often observed that students are very inefficient in their work. They frequently use methods of working that are unproductive and slow. Some examples:

- Students do not know how to touch-type. Instead of taking the relatively limited time to learn to do it, they waste many hours per week on slow typing and typing errors.
- Students frequently do not know how to use advanced features in the text editor such as the interface to the version-control system, the interface to the Lisp system, etc. Again instead of taking a short time to learn, they waste much more time.
- Students do not know how to use a debugger. Instead, they waste time debugging programs with trace output.
- etc.

As it turns out, this practice is not restricted to students, but is also common in the software industry.

But why do people deliberately waste time when there are much more efficient ways of working? This is a very good question. In fact, it is such a good question that I decided to ask a well-known professor of psychology at one of the top universities on the east coast of the USA. What she told me was no doubt a simplification so that a layman like myself could understand it. Despite such simplifications, her explanation both gave me a much better understanding of the phenomenon and some ideas about how compensate for it.

She told me that (with respect to this phenomenon) people can be roughly divided into two categories that she called *perfection-oriented* and *performance-oriented*.

The people in the category perfection-oriented have a natural intellectual curiosity. They are constantly searching for better ways of doing things, new methods, new tools. They search for perfection, but they take pleasure in the search itself, knowing perfectly well that perfection can not be accomplished. To the people in this category, failure is a normal part of the strive for perfection. In fact, failure gives a deeper understanding of *why* a particular path was unsuccessful, making it possible to avoid similar paths in the future.

The people in the category performance-oriented on the contrary, do not at all strive for perfection. Instead they have a need to achieve performance *immediately*. Such performance leaves no time for intellectual curiosity. Instead, techniques already known to them must be applied to solve problems. To these people, failure is a disaster whose sole feature is to harm instant performance. Similarly, learning represents the possibility of failure and must thus be avoided if possible. To the people in this category, knowledge in other people also represents a threat. As long as everybody around them use tools, techniques, and methods that they themselves know, they can count on outperforming these other people. But when the people around them start learning different, perhaps better, ways, they must defend themselves. Other people having other knowledge might require learning to keep up with performance, and learning, as we pointed out, increases the risk of failure. One possibility for these people is to discredit other people's knowledge. If done well, it would eliminate the need for the extra effort to learn, which would fit very well with their objectives.

This model of learning also explains other surprising behavior that I frequently observe. I have seen novices in software development with knowledge of a single programming language explain to experienced expert developers why their choice of programming language was a particularly bad one. In one case, I talked to a student of computer science who told me why a particular programming language was bad. In fact he told me it was so bad that he had moved to a different university in order to avoid courses that used that particular language. When asked, he admitted he had never written a single program in that language. He simply did not know what he was talking about. And he was willing to fight for it. With respect to programming languages, negative opinions about a language that a person does not know, are usually based on very superficial aspects of it. To people obsessed with performance lack of such in a programming language is a favorite reason to advocate its eradication (even though performance is not a quality of a language, but of a particular implementation).

As the reader has probably already guessed, my surprising observations concern mostly performance-oriented people. The above discussion is obviously a simplification. In particular, a person can be in one category with respect to a particular domain, and in the other with respect to another domain. Thus, I have seen professors in mathematics who were obviously perfection-oriented with respect to mathematics, be firmly in the performance-oriented category with respect to the efficient use of (say) word processors. It is almost a surrealistic experience to see a person in one situation full of intellectual curiosity and wanting to know everything about everything, and in another situation argue why you should not use a particular method that he himself does not know anything about, for reasons that are obviously totally artificial.

Thus, what I have observed is not only what one might expect, i.e., some reluctance to learning new tools and methods, but a kind of reaction orders of magnitude stronger than I had expected. I have observed that people ignorant in a particular domain, or not knowing a particular tool or technique, would go to great trouble to explain why knowing this domain, tool, or technique, would be a complete waste of time. Usually these explanations were based on erroneous ideas of what it represented. To make things worse, they were perfectly willing to present their erroneous arguments to the very experts in the field in question.

Similarly, I have heard people argue against a tool that they ignore based on the fact that it can do *too much*. Too much functionality in a tools is a problem only if unneeded or unwanted functionality somehow makes it harder to use the needed and wanted parts. I have heard people argue about the amount of memory a particular tool requires, whereas the additional memory required might represent a cost equivalent to a few hours of work at most. A favorite idea is to label a particular tool with a name suggesting what it ought to be doing, and then arguing that it is doing *more* than that. For instance, a text editor that is capable of automatic indentation would be accused of being a "kitchen-sink" tool because after all it does much more than allowing the user to just edit text.

Needless to say, these people make complete fools of themselves. But that does not seem to bother them in any way whatsoever.

It is hard to overestimate the strength of this phenomenon. I myself recently discovered a marvelous feature in a programming language that I had purposely avoided for the past 10 years, simply because 10 years ago, a colleague (who did not know the feature) explained to me that it was no good. We were both victims of our own minds. My colleague because he obviously needed to defend that he had made a different choice, and myself because I subconsciously found it very appealing to be able to brush off the feature as useless and thus not having to learn it. It is hard to overestimate the wasted time I have put in during the past 10 years due to considerably lower productivity than I could have had, had I realized at the time what I now know about human psychology.

It is my hope that this explanation of a common phenomenon makes it possible for students to reflect upon their own motivations, and that it ultimately makes all students perfection-oriented.

6. Syntax of programming languages

In this section, we will try to explain what *syntax* means and how it is important to programming.

6.1. Possible definitions of syntax

The definition of syntax is not as clear as one might think.

For instance, one possible definition of syntax would be "the rules that create valid programs." That definition would cover a large part of what is usually considered semantics as well. In addition, syntax rules with this definition would be undecidable, making it impossible for the compiler to verify syntax. Usually when we think of syntax, we mean something that can be statically verified by the compiler.

Another possible definition of syntax would be "the context-free grammar that generates potentially valid programs." This is the usual definition of syntax in traditional languages such as C and Java. As we shall see, this definition is not adequate for Common Lisp.

Usually, in traditional languages, each *special form* has its own rule or rules within the grammar of the language. For instance, a `while` statement in C would have the following grammar rule:

```
::= while (  )  ;
```

which means that in order to write a `while` statement in C, you have to write the word `while` followed by an opening parenthesis, followed by an expression, followed by a closing parenthesis, followed by a statement followed by a semicolon.

6.2. Syntax in Common Lisp

It is important to realize that Common Lisp has a two-level syntax. The lowest level corresponds to the second definition above, but in Common Lisp, it is trivial. In fact, it is so trivial that we can state the essence of it here:

- A form is either an atom or a sequence of forms surrounded by parentheses.
- An atom is either a symbol, a number, a character, a string, or a quoted form.

This syntax is so general that it allows programs that are obviously statically incorrect. In particular, it does not prevent numerous illegal versions of special forms.

The low-level syntax of Common Lisp is implemented by a function called `read`. That function is the *parser* for Common Lisp. Its only purpose is to verify the low-level syntax, and transform the text of the program into an *internal representation*. That internal representation uses standard Common Lisp data types such as symbols, numbers, and lists. In particular, a sequence of forms surrounded by parentheses is transformed into a Common Lisp list of the transformed elements of the sequence.

Not only does the `read` function use standard Common Lisp types for its internal representation. That internal representation is *standardized by the language definition* and available to the programmer. This is so important that every Common Lisp programmer, when looking at some code, thinks about lists and atoms, rather than about sequences of tokens or characters.

The second-level syntax of Common Lisp has rules about the format of the internal representation. For instance, the `if` special form has the following format:

```
(if    )
```

or

```
(if   )
```

where `test-form`, `then-form`, and `else-form` are forms. But one should not think of this rule as being stated as sequences of tokens. Rather, one should think: "An `if` special form is a list with three or four elements. The first element is the symbol `if`, the second, third and optionally the forth are all forms."

The second-level syntax rules are checked by the compiler *after* the `read` function has transformed the program into the internal form. That part of the compiler is not restricted to context-free grammars or any other restrictions on sequences of symbols. It is free to traverse the internal representation in any order it needs, and as many times as necessary in order to verify syntax and generate code.

Perhaps the greatest advantage of the two-level syntax, and in particular of the standardized internal representation for programs, is that it makes it possible to have a powerful macro facility.

In ordinary programming languages such as C, the macro facility is very primitive, in that it replaces text by other text. This fact makes it possible for errors in definitions and uses of macros to generate mysterious defects that are extremely hard to find and debug.

In Common Lisp, on the other hand, the macro facility replaces *forms* by other forms. This fact makes it possible for the macro facility to be a useful mechanism for *extending the syntax* (i.e. the second-level syntax) of the language. Extending the syntax of the language is necessary in order to build an application-specific language on top of the base language.

In fact, large parts of a typical Common Lisp implementation are implemented as macros that translate special forms to simpler special forms.

7. Autonomous functions

We have chosen the term *autonomous function* for a function that accomplishes its work using only:

- values of arguments and local variables computed from these arguments,
- globally defined constants, and
- globally defined autonomous functions that are assumed to be nearly constant.

Autonomous functions are much easier to test and debug than other functions, since they yield the same result every time they are executed. They are not influenced by changes in the global environment such as values of global variables).

Notice that the definition is recursive, so that autonomous functions can be built as long as only other autonomous functions are used in their implementation.

This fact greatly simplifies bottom-up programming, in which more and more complex functions are created out of simpler, lower-level functions. Keeping these functions autonomous makes it possible to test and debug very complicated functions in a very efficient way.

8. Side-effect free functions

Side effects are operations that change the global state of a computation. Formally, all assignments and all input/output operations are considered side-effects. The term *functional programming* (or more accurately *purely functional programming*) refers to a style of programming where functions have no side effects.

While it is impossible to write any real applications without any side effects (you could never write anything to the user, and you could never save anything to disk), functional programming is still a very important way of programming large parts of an application.

We can get most of the benefits of functional programming while still allowing some kinds of side effects, for instance:

- input/output that does not effect the global state of the system, for instance output to the screen,
- assignments to local variables that do not survive the execution of the function in which they are located.

In other words, we want the execution of a function to have *no effect on the global state of the system.* If, in addition, the function is autonomous, we can be sure that repeated executions of the same function with the same arguments gives exactly the same result.

If the function is not without side effects, the following can happen:

```
> (setq *l* (list 1 2 3 4 5))
(1 2 3 4 5)
> (defun fun (l)
    (setq *l* (cdr l))
    (length l))
fun
> (fun *l*)
5
> (fun *l*)
4
> (fun *l*)
3
```

Notice that `fun` is by definition autonomous, but we lose because it has a side effect on the global variable `*l*`. If instead, we had written: ¿ (defun fun (l) (setq *l* (cdr l)) (length l)) fun ¿ (fun '(1 2 3 4 5)) 5 ¿ (fun '(1 2 3 4 5)) 5 ¿ (fun '(1 2 3 4 5)) 5

9. Indenting Common Lisp

9.1. Introduction

The term *indentation* refers to the amount of white space at the beginning of lines of code. But here we use the term in a slightly more general way. We use it to mean all rules for the physical layout of a program, such as when to use newlines, how many semicolons to use in comments, etc. It is very important to indent Common Lisp programs correctly; much more important than to indent C or Java programs. The reason is that experienced Lisp programmers read and understand Lisp code based on indentation as opposed to some other way, such as counting parentheses.

But the programmer cannot be trusted to indent correctly. Therefore, it is equally important that the text editor (usually Emacs) be taught how to indent properly.

Notice that indentation rules are *not* a matter of personal taste, but a matter of *shared culture*. The rules for indenting Lisp programs have evolved over a number of years as a kind of standard in the Lisp community. It is *not* up to the individual student to invent his or her own indentation rules. The argument is one of maintainability. A program is usually maintained by people other than those who wrote it. Standardized rules for indentation play an important rôle in facilitating maintenance, in particular of Lisp programs.

In this section, we give the most common rules for indenting Lisp programs, together with the Emacs expressions that accomplish them.

9.2. General rules

9.2.1. Top-level functions

All top-level function definitions start in column 1. This is important, since Emacs uses that information to search for the beginning of function definitions.

9.2.2. Closing parentheses

Closing parentheses are never preceded by newlines. Thus, do *not* write like this:

```
;;; bad
(defun f (x)
  (when (
```

Instead write like this:

```
;;; good
(defun f (x)
  (when (
```

The first method may seem better to some people, since it is easier to match parentheses, but as we have already mentioned, readers of Lisp programs do not match parentheses, but use indentation.

The first method also requires more lines for the same code. In general, it is good to keep the number of lines down, so that more code will fit on a page or a screen; subject to the indentation rules, of course.

9.2.3. Amount of indentation

Unless more specific rules state otherwise, the level of indentation should be relatively small. We use two characters. Thus, do not write like this:

```
;;; bad
(defun f (x)
    (when (
```

Instead write like this:

```
;;; good
(defun f (x)
  (when (
```

The reason for the modest indentation is to lower the probability of lines sticking too far out to the right, which in turn would require more newlines and thus more lines in the program. Again, we try to keep the number of lines down.

9.3. Comments

9.3.1. Number of semicolons

Comments in Common Lisp start with a semicolon and end at the end of a line. Over the years, a style has developed that uses between one and four semicolons, according to the scope of the comments.

A single semicolon is used for a comment concerning a single line of code, and located on the same line as the code, for instance:

```
(if (
```

Two semicolons are used for a comment that cover several lines of code. The comment line is lined up the the code lines, and precedes them, like this:

```
(when (
```

Three semicolons are used for comments that explain a function. Such comments always start in column 1, like this:

```
;;; Compute the amount of space between symbols
;;; as a list of floating point values.
(defun compute-spaces (symbols)
  (mapcar #'compute-single-space symbols (cdr symbols)))
```

9.3.2. Content of comments

As usual, we try to be brief, without losing the information contents. For comments to function definitions, a good idea is to use the imperative form of the verb. This way you can avoid redundant expressions such as "this function". Do not write like this:

```
;;; This function computes the space between symbols
;;; as a list of floating point values
(defun compute-spaces (symbols)
  (mapcar #'compute-single-space symbols (cdr symbols)))
```

Instead, write like this:

```
;;; Compute the amount of space between symbols
;;; as a list of floating point values.
(defun compute-spaces (symbols)
  (mapcar #'compute-single-space symbols (cdr symbols)))
```

9.4. Indenting special forms

Each special form has its own indentation rule. Here, we give a list of the most common ones.

9.4.1. Indenting the if special form

The if special form in Common Lisp takes exactly three subexpressions. The subexpressions should be lined up, like this:

```
(if (= (f x) 4)
    (top-level x)
    (g x))
```

This indentation is accomplished by the following Emacs expression that you can put in your .emacs file:

```
(put 'if 'lisp-indent-function nil)
```

9.4.2. Indenting the when and unless special forms

These forms both take an arbitrary number of subexpressions, the first of which is distinguished as the test. We write the test on the same line as the name of the special form, and all the other forms on separate lines, indented by two positions, like this:

```
(when (= (f x) 4)
  (setf *level-number* 0)
  (unless *do-not-reinitialize*
    (reinitialize-global-information x)
    (reinitialize-local-information))
  (top-level x))
```

This indentation is accomplished by the following Emacs expression that you can put in your .emacs file:

```
(put 'when 'lisp-indent-function 1)
(put 'unless 'lisp-indent-function 1)
```

9.4.3. Indenting the let and let* special forms

These forms both take an arbitrary number of subforms. The first is a list of variable initializations, and the rest are the body of the special form. The variable initializations start on the same line as the name of the special form, and the initializations is lined up. The body is indented by two characters like this:

```
(let* ((symbols (mapcar #'compute-symbol l))
       (spaces (mapcar #'compute-space symbols (cdr symbols))))
  (when (verify-spacing symbols spaces)
    (make-spacing permanent spaces)))
```

This indentation is accomplished by default in emacs.

9.4.4. Indenting the do and do* special forms

These forms both take an arbitrary number of subforms, the first two of which are distinguished. The first one contains the loop variables and the second is a loop termination form. The rest are the body of the special form. Here is an example:

```
(do ((i 1 (1+ i))
     (j (length l) (/ j 2)))
    ((= j 0) i)
  (iterate i j)
  (when (= (f x) 4)
    (setf *level-number* 0)
    (top-level x)))
```

This indentation is accomplished by the following Emacs expression that you can put in your .emacs file:

```
(put 'do 'lisp-indent-function 2)
(put 'do* 'lisp-indent-function 2)
```

10. Top-down programming

Top-down programming is the opposite of bottom-up programming. It refers to a style of programming where an application is constructed starting with a high-level description of what it is supposed to do, and breaking the specification down into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used.

10.1. Disadvantages of top-down programming

Top-down programming complicates testing. Noting executable exists until the very late in the development, so in order to test what has been done so far, one must write stubs .

Furthermore, top-down programming tends to generate modules that are very specific to the application that is being written, thus not very reusable.

But the main disadvantage of top-down programming is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over time. When that happens, there is a great risk that large parts of the application need to be rewritten.

10.2. How does top-down programming work?

Top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures. Typically, the high-level specification of the system states functionality. This high-level description is then refined to be a sequence or a loop of simpler functions or procedures, that are then themselves refined, etc.

In this style of programming, there is a great risk that implementation details of many data structures have to be shared between modules, and thus globally exposed. This in turn makes it tempting for other modules to use these implementation details, thereby creating unwanted dependencies between different parts of the application.

11. Bottom-up programming

11.1. What is bottom-up programming

Bottom-up programming is the opposite of top-down programming. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written.

11.2. Advantages of bottom-up programming

Bottom-up programming has several advantages over top-down programming .

Testing is simplified since no stubs are needed. While it might be necessary to write test functions, these are simpler to write than stubs, and sometimes not necessary at all, in particular if one uses an interactive programming environment such as Common Lisp or GDB.

Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down. In fact, one can argue that the purpose bottom-up programming is to create an application-specific language . Such a language is suitable for implementing an entire class of applications, not only the one that is to be written. This fact greatly simplifies maintenance, in particular adding new features to the application. It also makes it possible to delay the final decision concerning the exact functionality of the application. Being able to delay this decision makes it less likely that the client has changed his or her mind between the establishment of the specifications of the application and its implementation.

11.3. How does bottom-up programming work?

In a language such as C or Java, bottom-up programming takes the form of constructing abstract data types from primitives of the language or from existing abstract data types.

In Common Lisp, in addition to constructing abstract data types, it is common to build *functions* bottom-up from simpler functions, and to use macros to construct new *special forms* from simpler ones.

One may ask why it is not possible to construct functions and special forms bottom-up in other languages than Common Lisp. Constructing functions bottom-up requires a way of passing complicated arguments between functions. Common Lisp uses *lists* for such argument passing. Lists are flexible standardized data structures in the language.

In other languages, data structures would have to be defined for such parameter passing only, making it more like an abstract data type than just a function. With respect to special forms, only the two-level syntax of Common Lisp allows a flexible enough macro facility for bottom-up programming of special forms.

12. Building specialized languages

A programming language is roughly made up of built-in procedures and functions, and of special forms. The built-in procedures and functions are typically for arithmetic, list manipulation, input/output, etc. Special forms make up the syntax of the language. Typical special forms are conditionals such as `if`, and loop constructs such as `while` and `for`.

In addition, most programming languages have a way of constructing new functions and procedures out of existing ones. These new procedures work like the built-in ones, except that in most languages the syntax is different. For instance, in C you could write `a + b` for applying the `+`function to `a` and `b`. However, if you write your own function, say `f`, you cannot write `a + b`. Instead, you have to write `f(a, b)`.

In some cases, extensive *libraries* exist, where someone has created a collection of procedures and functions useful for some specific purpose, say graphics. Such libraries represent a way to *customize* the language to a particular class of applications. In other words, the language is transformed from a general-purpose one, to an application-specific language.

But, whether a language is convenient to use for a particular class of applications depends both on the procedures and functions (whether built-in or in libraries), and the special forms. Unfortunately, most languages do not allow the user to create new special forms out of existing ones.

Common Lisp allows the user to extend the language both with procedures and functions, and with special forms. This feature makes it possible to transform Common Lisp to an application-specific language in all possible respects.

13. Object-oriented programming

13.1. Introduction

In the past couple of years, object-oriented programming has become a buzzword roughly equivalent to "good." In this section, we give a more realistic presentation of object-oriented methods, and what they stand for.

Object-oriented programming was invented to solve certain problems of modularity and reusability that occur when traditional programming languages such as C are used to write applications. This fact makes it hard for a non-experiences programmer to appreciate object-oriented techniques.

To avoid being influenced by specific programming languages and their particularities, we start by giving a very general presentation of the ideas behind object-oriented programming. We then discuss how these general techniques map to specific programming languages such as Java or CLOS (the Common Lisp Object System).

13.2. General terminology

13.2.1. Object

An *object* defined to be a value that can be assigned to variables, supplied as arguments to functions, and returned as values of function calls. An object has an *identity* so that two values can be compared to test whether they are in fact the *identical* object. If uniform reference semantics is used, then assignment of objects preserve this identity. Usually, but not always, an object is implemented as a pointer to a block of memory that is allocated for the object when the object is created. But simple objects such as integers and floating-point numbers may not have any allocated memory associated with them. Instead, they are fully contained in the value itself. Identity and uniform reference semantics can still be preserved, provided that no side effects are possible to such objects.

13.2.2. Abstract data type

An *abstract data type* (or *type*, for short) is a set of objects. Types are not necessarily mutually exclusive, so that an object can be of several types. In particular, a type can be a subset of another type, so that all objects of type `automobile` are also objects of type `vehicle`.

13.2.3. Operation

An *operation* is a function or a procedure whose arguments are objects. Each possible argument is of a certain type. For instance, `mount-engine` might be an operation that takes two arguments, an object of type `motor-vehicle` and an object of type `engine`, and has the side effect of altering the engine of the motor vehicle. Similarly, `push` might be an operation that takes two arguments, an object of type `stack` and an object of arbitrary type, and has the side effect of inserting the second argument on the top of the stack.

13.2.4. Protocol

A *protocol* is a collection of operations that are usually needed simultaneously. Typically, at least one type is common between all the operations in the protocol. An example of a protocol would be *drawing graphics objects*. Such a protocol would contain operations such as `draw`, `erase`, and `move` using objects of type `window`, `pixmap`, `circle`, `rectangle`, etc.

13.2.5. Class

A *class* describes the implementation details of a collection of objects, such as the number and names of the fields (or slots). A *type* is typically made up of a collection of classes that may or may not be related by *inheritance*.

An object is said to be a *direct instance* of exactly one class, and that class is called the *class of* the object.

13.2.6. Inheritance

Classes are organized in a hierarchy that can be a directed acyclic graph (multiple inheritance) or a tree (single inheritance). An object is the *instance* of a class C if and only if it is either a direct instance of C, or an instance of a class that *inherits* from C (or, equivalently, is *derived from* C).

If a type contains the class C, then it also contains all classes derived from C.

13.3. Object-oriented programming in CLOS

13.3.1. Method

A method in CLOS is a partial implementation of an operation. It is partial, in the sense that it implements the operation for a particular set of classes of the arguments. For the operation to be completely implemented, methods are needed that cover all possible classes that make up all possible types possible of the arguments.

The methods that make up the implementation of an operation can be physically distributed. They are recognized as belonging together because they have the same name.

13.3.2. Generic functions

A *generic function* is the complete implementation of an operation. It is used as an ordinary function. A particular partial implementation, a method, is chosen according to the classes of the arguments.

13.3.3. Generic dispatch

The mechanism that selects a method of a generic function according to the classes of the arguments is called *generic dispatch*. CLOS has *multiple dispatch*, which means that the generic dispatch mechanism can take into account *several* arguments in order to decide which method to use.

13.3.4. Multiple inheritance

CLOS supports multiple inheritance, which means that a class can inherit from several parent classes.

13.4. Object-oriented programming in Java and other languages

In Java, C++, Smalltalk, Eiffel, Simula, and other object-oriented languages, things work differently from CLOS.

13.4.1. Static typing

Some of these languages are *statically typed*, which means that the user has to declare what kind of objects can be the value of a particular variable or parameters. Roughly speaking, the declared type of a such variables or parameters must be the name of a class. As a consequence, an *abstract data type* must also be a *class*. This is accomplished by always creating a class (potentially without any instances), from which all other classes that make up the type are derived. The name of the root class of such a hierarchy is usually the same as the name of the type.

13.4.2. Restrictions on operations

All of these languages use only *single dispatch* which means that a partial implementation of an operation can be selected based on only one (the first) argument. In fact, all of these languages threat the first argument so differently that there is a special syntax for it. Whereas in CLOS, one would say:

```
(mount-engine *my-car* *the-engine*)
```

In these languages, one would say something like:

```
my_car.mount_engine(the_engine)
```

if dispatch is wanted on the type of car, and

```
my_engine.mount_engine(my_car)
```

if dispatch is wanted on the type of engine.

That special syntax makes it impossible to pass a generic function (here `mount_engine`) as an argument to another function.

Since methods are so closely related to one particular class, in these languages methods are in fact physically written inside the declaration of the class itself, further cementing this relationship.

13.4.3. Single or multiple inheritance

Some of these languages support only single inheritance, such as Smalltalk. C++ supports multiple inheritance. Java, although supporting only single inheritance, has a feature called *interfaces* which makes it possible to obtain most of the features of multiple inheritance.

Glossary

abstract data type We use the term abstract data type differently from the usual definition used in computer science research. Here, and abstract data type is a type with a well-defined interface and semantics.

Examples of abstract data types are: stack. queue, union-find set, priority queue, etc.

Some data types don't have well-defined interfaces, but are useful for implementing abstract data types. We call them implementation types.

buffer An array of elements. When used to implement a container, usually dynamically resizeable. A buffer is not an abstract data type, but an implementation type.

container An abstract data type whose sole purpose is to contain other objects. Different containers differ in the way the elements are accessed. Examples of containers are: set, stack, queue, double-ended queue, priority queue, dictionary, etc.

defect Popularly known as a *bug*. A programming error that causes the program to *fail*, i.e. not accomplish its nominal task.

delegation A technique that consists of using a more powerful abstract data type to implement a less powerful one, by using a subset of the operations on the more powerful data structure.

double-ended queue A container holding a sequence of elements and allowing you to insert and delete elements both at the beginning and at the end of the sequence.

exception 1. An unusual situation that occurs as a result of the execution of parts of a program, such as attempting to open a file that does not exist, or attempting to allocate memory where non is available.

2. A mechanism in certain programming languages designed specifically to handle exceptional situations.

failure A situation where a program is in a state that was not foreseen by the programmer. A program in such a state has a *defect* or is said to be *defective*.

A failure may or may not be *detected* by the program itself. If, undetected, the failure just causes the program to do the wrong thing.

generic pointer A pointer of type `void *`. It is called *generic* because it can point to any object. This is useful for implementing *containers*.

header object An object sometimes required in order to preserve uniform reference semantics.

implementation type Implementation types are types that are not well-defined enough that they can be considered abstract data types.

Examples of implementation types are: list, doubly-linked list, circular list, binary tree, table.

list Same as a linked list. Can be simply or doubly linked, but usually simply linked if not explicitly known to be doubly linked.

A linked list is not an abstract data type, but an implementation type. There is therefore no point in trying to write a module named `list` with a well-defined interface. Usually, a linked list will be defined like this:

```
typedef struct cell *list;

struct cell
{
  void *element;
  list next;
};
```

nonlocal goto A *goto*-instruction that jumps to a place *outside* the procedure in which it is located.

Nonlocal gotos can be used as a primitive mechanism for handling *exceptions*.

predicate A function that that returns a boolean value. Usually, predicates are side-effect free, and are used to test the state of some kind of data structure.

queue A container holding a sequence of elements and allowing you to insert elements at one end of the sequence and delete elements from the other end of the sequence.

scaffolding Pieces of programs whose sole purpose is to aid in testing and debugging. Scaffolding is not part of the application itself. Scaffolding may take the form of *stubs*, or of *test functions*.

set An unordered collection of items. Sets are very hard to implement efficiently in general. It is therefore common to find special versions of sets, such as ordered sets, sets that only support a subset of the usual set operations, etc.

side effect An operation that changes the global state of the program. It is good practice to avoid side effects whenever practical, as it makes the program easier to maintain and to test.

stack A container holding a sequence of elements and allowing you to insert elements at one end of the sequence and delete elements from the same end of the sequence.

stubs Pieces of programs, usually functions or procedures, that provide the correct *interface* but not the correct *implementation* of some other piece of program, typically a function or an *abstract data structure*. A stub makes it possible to test certain aspects of a high-level function or abstract data type without having implemented all of the lower-level functions or abstract data types that it uses. Stubs are necessary when top-down programming is used.

test functions Functions whose sole purpose is to be clients of other functions or abstract data types, in order to test the latter. Test functions are not part of the application, but exist only to test other parts of the application before the real clients are written. Test functions are necessary when bottom-up programming is used. However, sometimes an *interactive programming environment* such as Common Lisp or *GDB* can replace parts or all of the test functions necessary.