Computer Science Buzz-words

From Wikipedia, the free encyclopedia.

Compiled by Tore Haug-Warberg
Department of Chemical Engineering, NTNU
September 17th 2004

Table of contents

Operator overloading

In computer programming, operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism in which some or all of operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of its arguments. Operators need not always be symbols.

Operator overloading is usually only a syntactic sugar, and it can be easily emulated by function calls:

with operator overloading
a + b × c

without operator overloading
our_new_type_add (a, our_new_type_multiply (b,c))

Only in case when operators can be called implicitly they are of some use other than aesthetics. This is the case with Ruby operator to_s, which returns a string representation of an object and with operators in PostgreSQL, where mathematical transformations can be defined on operators and PostgreSQL may use many optimalizations to expressions that use them.

Polymorphism

In computer science, polymorphism is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations.

The concept of polymorphism applies to functions as well as types:
A function that can evaluate to and be applied to values of different types is known as a polymorphic function. A datatype that contains elements of an unspecified type is known as a polymorphic datatype.

There are two fundamentally different kinds of polymorphism:
If the range of actual types that can be used is finite and the combinations must be specified individually prior to use, we call it ad-hoc polymorphism.
If all code is written without mention of any specific type and thus can be used transparently with any number of new types, we call it parametric polymorphism.

Programming using the latter kind is called generic programming.

Polymorphism gained most of its momentum when object-oriented programming became popular.

Parametric polymorphism

Using parametric polymorphism, a function can been written generically so that it can deal equally well with objects of various types. For example, a function append that joins two lists can be constructed so that it does not depend on one particular type of list: it can append lists of integers, lists of real numbers, lists of strings, and so on. Let a denote the type of elements in the lists. Then append can be typed [a] × [a] → [a], where [a] denotes a list of elements of type a. We say that append is parameterized by a. (Note that since there is only one type parameter, the function cannot be applied to just any pair of lists: they must consist of the same type of elements.)

Parametric polymorphism was the first type of polymorphism developed, first identified by Christopher Strachey in 1967. It was also the first type of polymorphism to appear in an actual programming language, ML in 1976. It exists today in Standard ML, OCaml, Haskell, and others. Some argue that templates should be considered an example of parametric polymorphism, though they rely on macros to generate specific code rather than actually reusing generic code (resulting in code bloat).

Parametric polymorphism is a way to make a language more expressible, while still maintaining full static type-safety. It is thus irrelevant in dynamically typed languages, since they by definition lack static type-safety. However, any dynamically typed function f that takes n arguments can be given a static typed using parametric polymorphism: f : p1 × ... × pn → r, where p1 ... pn and r are type parameters. Of course, this type is completely insubstantial and thus essentially useless.

Subtyping polymorphism

Some languages employ the idea of a subtype to restrict the range types that can be used in a particular case of parametric polymorphism. Hence, subtyping polymorphism allows a

function to be written so that it takes a certain type of object T, but will also work correctly if passed an object that belongs to a type S that is a subtype of T. This is sometimes written S <: T. Conversely, we say that T is a supertype of S—written T :> S.

For example, given the types Number, Integer, and Fractional, such that Number :> Integer and Number :> Fractional, a function could be written that takes any kind of Number, and works equally well whether an Integer or Fractional is passed. (This particular kind of type hierarchy is known—especially in the context of Scheme—as a numerical tower, and usually contains a lot more types.)

In object-oriented programming this is implemented with subclassing (inheritance), and is also known as late binding.

See also:
Polymorphism in object-oriented programming,
Liskov substitution principle.

Ad-hoc polymorphism

Ad-hoc polymorphism usually refers to simple overloading (see function overloading), but sometimes automatic type conversion, known as coercion, is also considered to be a kind of ad-hoc polymorphism (see the example section below). Common to these two types is the fact that the programmer has to specify exactly what types are to be usable with the polymorphic function.

The name refers to the manner in which this kind of polymorphism is typically introduced: "Oh, hey, let's make the + operator work on strings, too!" Some argue that ad-hoc polymorphism is not polymorphism in a meaningful computer science sense at all—that it is just syntactic sugar for calling append_integer, append_string, etc., manually. One way to see it is that

> to the user, there appears to be only one function, but one that takes different types of input and is thus type polymorphic; on the other hand,
> to the author, there are several functions that need to be written—one for each type of input—so there's essentially no polymorphism.

Overloading

Overloading allows multiple functions taking different types to be defined with the same name; the compiler or interpreter automatically calls the right one. This way, functions appending lists of integers, lists of strings, lists of real numbers, and so on could be written, and all be called append—and the right append function would be called based on the type of lists being appended. This differs from parametric polymorphism, in which the function would need to be written generically, to work with any kind of list. Using overloading, it is possible to have a function perform two completely different things based on the type of input passed to it; this is not possible with parametric polymorphism.

This type of polymorphism is common in object-oriented programming languages, many of which allow operators to be overloaded in a manner similar to functions (see operator overloading). It is also used extensively in the purely functional programming language Haskell. Many languages lacking ad-hoc polymorphism suffer from long-winded names such as print_int, print_string, etc. (see Objective Caml).

Coercion

Due to a concept known as coercion, a function can become polymorphic without being initially designed for it. Let f be a function that takes an argument of type T, and S be a type that can be automatically converted to T. Then f can be said to be polymorphic with respect to S and T.

Some languages (e.g., C, Java), provide a fixed set of conversion rules, while others (e.g., C++) allow new conversion rules to be defined by the programmer. While calling C "polymorphic" is perhaps stretching it, the facility for automatic conversion (i.e., casting operators) found in C++ adds a whole new class of polymorphism to the language.

Example

This example aims to illustrates the three different kinds of polymorphism described in this article. Though overloading an originally arithmetic operator to do a wide variety of things in this way may not be the most clear-cut example, it allows some subtle points to be made. In practice, the different types of polymorphism are not generally mixed up as much as they are here.

Imagine, if you will, an operator + that may be used in the following ways:
1 + 2 ⇒ 3
3.14 + 0.0015 ⇒ 3.1415
1 + 3.7 ⇒ 4.7
[1, 2, 3] + [4, 5, 6] ⇒ [1, 2, 3, 4, 5, 6]
[true, false] + [false, true] ⇒ [true, false, false, true]
"foo" + "bar" ⇒ "foobar"

Overloading

To handle these six function calls, four different pieces of code are needed—or three, if strings are considered to be lists of characters:
In the first case, integer addition is invoked.
In the second and third cases, floating-point addition is invoked.
In the fourth and fifth cases, list concatenation is invoked.
In last case, string concatenation is invoked, unless this too is handled as list concatenation.
Thus, the name + actually refers to three or four completely different functions. This is an example of overloading.

Coercion

As we've seen, there's one function for adding two integers and one for adding two floating-point numbers in this hypothetical programming environment, but note that there is no function for adding an integer to a floating-point number. The reason why we can still do this is that when the compiler/interpreter finds a function call f(a1, a2, ...) that no existing function named f can handle, it starts to look for ways to convert the arguments into different types in order to make the call conform to the signature of one of the functions named f. This is called coercion. Both coercion and overloading are kinds of ad-hoc polymorphism.

In our case, since any integer can be converted into a floating-point number without loss of precision, 1 is converted into 1.0 and floating-point addition is invoked. There was really only one reasonable outcome in this case, because a floating-point number cannot generally be

converted into an integer, so integer addition could not have been used; but significantly more complex, subtle, and ambiguous situations can occur in, e.g., C++.

Parametric polymorphism

Finally, the reason why we can concatenate both lists of integers, lists of booleans, and lists of characters, is that the function for list concatenation was written without any regard to the type of elements stored in the lists. This is an example of parametric polymorphism. If you wanted to, you could make up a thousand different new types of lists, and the generic list concatenation function would happily and without requiring any augmentation accept instances of them all.

It can be argued, however, that this polymorphism is not really a property of the function per se; that if the function is polymorphic, it is due to the fact that the list datatype is polymorphic. This is true—to an extent, at least—but it is important to note that the function could just as well have been defined to take as a second argument an element to append to the list, instead of another list to concatenate to the first. If this were the case, the function would indisputably be parametrically polymorphic, because it could then not know anything about its second argument

Polymorphism, in computer underground terms, also refers to polymorphic code, computer code that mutates for each new time it is executed. This is sometimes used by computer viruses, computer worms and shellcodes to hide the presence of their decryption engines.

Procedural (imperativ) programming

Procedural programming is a method (a programming paradigm) of computer programming based upon the concept of the unit and scope (the data viewing range of an executable code statement). A procedural program is composed of one or more units or modules--either user coded or provided in a code library; each module is composed of one or more procedures, also called a function, routine, subroutine, or method, depending on programming language. It is possible for a procedural program to have multiple levels or scopes, with procedures defined inside other procedures. Each scope can contain variables which cannot be seen in outer scopes.

Procedural programming offers many benefits over simple sequential programming: Procedural programming code is easier to read and more maintainable; Procedural code is more flexible; Procedural programming allows for the easier practice of good program design.

See also:
Structured programming
Procedural programming language
Programming language
list of programming languages

Compare with: functional programming, object-oriented programming

External link
Procedural programming languages

Declarative programming

Declarative programming is an approach to computer programming that takes a different approach from traditional imperative programming in Fortran, C++ or Java. Whereas imperative programming gives the computer a list of instructions to execute in a particular order, declarative programming describes to the computer a set of conditions and lets the computer figure out how to satisfy them. Declarative programming includes both functional programming and logic programming.

Declarative languages describe relationships between variables in terms of functions or inference rules. The language executor (an interpreter or compiler) applies a fixed algorithm to these relations to produce a result.

Examples of declarative programming languages include Miranda, Prolog and SQL.

Declarative programming languages are extensively used in solving artificial intelligence and constraint-satisfaction problems.

See also: 4GL, constraint programming

Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions. In contrast to imperative programming, functional programming emphasizes the evaluation of functional expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values.

Introduction

The functions alluded to in the title are mathematical functions. Mathematical functions have great strengths in terms of flexibility and in terms of analysis. For example, if a function is known to be idempotent, then a call to a function which has itself as its argument, and which is known to have no side-effects, may be efficiently computed without multiple calls.

A function in this sense has zero or more parameters and a single return value. The parameters—or arguments, as they are sometimes called—are the inputs to the function, and the return value is the function's output. The definition of a function describes how the function is to be evaluated in terms of other functions. For example, the function f(x) = x2 + 2 is defined in terms of the power and addition functions. At some point, the language has to provide basic functions that require no further definition.

Functions can be manipulated in a variety of ways in a functional programming language. Functions are treated as first-class values, which is to say that functions can be parameters or inputs to other functions and can be the return values or outputs of a function. This allows functions like mapcar in LISP and map in Haskell that take both a function and a list as input and apply the input function to every element of the list. Functions can be named, as in other languages, or defined anonymously (sometimes during program execution) using a lambda abstraction and used as values in other functions. Functional languages also allow functions to be "curried". Currying is a technique for rewriting a function with multiple parameters as the composition of functions of one parameter. The curried function can be applied to just a subset of its parameters. The result is a function where the parameters in this subset of are now fixed as constants, and the values of the rest of the parameters are still unspecified. This new function can be applied to the remaining parameters to get the final function value. For example, a function add(x,y) = x + y can be curried so that the return value of add(2)—notice that there is no y parameter—will be an anonymous function that is equivalent to a function add2(y) = 2 + y. This new function has only one parameter and corresponds to adding 2 to a number. Again, this is possible only because functions are treated as first class values.

History

Lambda calculus could be considered the first functional programming language, though it was not designed to be executed on a computer. Lambda calculus is a model of computation designed by Alonzo Church in the 1930s that provides a very formal way to describe function evaluation. The first computer-based functional programming language was LISP, developed by John McCarthy while at the Massachusetts Institute of Technology in the late 1950s. While not a purely functional programming language, LISP did introduce most of the features now found in modern functional programming languages. Scheme was a later attempt to simplify and improve LISP. In the 1970s the language ML was created at the University of Edinburgh, and David Turner developed the language Miranda at the University of Kent. The language

Haskell was released in the late 1980s in an attempt to gather together many ideas in functional programming research.

Comparison with imperative programming

Functional programming can be contrasted with imperative programming. Functional programming appears to be missing several constructs often (though incorrectly) considered essential to an imperative language, such as C or Pascal. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked; memory allocation occurs to make space for the parameters and the return value, and assignment occurs to copy the parameters into this newly allocated space and to copy the return value back into the calling function. Both operations can only occur on function entry and exit, so side effects of function evaluation are eliminated. By disallowing side effects in functions, the language provides referential transparency. This ensures that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. Referential transparency greatly eases both the task of proving program correctness and the task of automatically identifying independent computations for parallel execution.

Looping, another imperative programming construct, is accomplished through the more general functional construct of recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. In fact, it can be proven that looping is equivalent to a special type of recursion called tail recursion. Recursion in functional programming can take many forms and is in general a more powerful technique than looping. For this reason, almost all imperative languages also support it (with FORTRAN 77 and COBOL as notable exceptions).

Functional programming languages

As detailed above, the oldest example of a functional language is Lisp. More recent examples include Scheme, ML, Haskell, Erlang, Clean, Q. "Pure" functional programs need no variables and side-effects, and are therefore automatically thread-safe, automatically verifiable (as long as any recursive cycle eventually stops) and have more such nice properties. Nested functions just pass their results back to the main function. Implementations of these languages usually make quite sophisticated use of stack manipulation, since it is used so commonly.

Functional programming often depends heavily on recursion. The Scheme programming language even requires certain types of recursion (tail recursion) to be recognized and automatically optimized by a compiler.

Furthermore, functional programming languages are likely to enforce referential transparency, which is the familiar notion that 'equals can be substituted for equals': if two expressions are defined to have equal values, then one can be substituted for the other in any larger expression without affecting the result of the computation. For example, in

```
z = f(sqrt(2), sqrt(2));
```

we can factor out sqrt(2) and write

```
s = sqrt(2);
z = f(s, s);
```

thus eliminating the extra evaluation of the square-root function.

As intuitive as it sounds, this is not always the case with imperative languages. A case in point is the C programming language's getchar() "function", which is strictly a function not of its arguments but of the contents of the input stream stdin and how much has already been read. Following the example above:

```
z = f(getchar(), getchar());
```

we cannot eliminate getchar() as we did for sqrt(2), because in C, "getchar()" might return two different values the two times it is called.

Hidden side-effects are in general the rule, rather than the exception, of traditional programming languages. Whenever a procedure reads a value from or writes a value to a global or shared variable, the potential exists for hidden side effects. This leakage of information across procedure boundaries in ways that are not explictly represented by function calls and definitions greatly increases the hidden complexity of programs written in conventional non-functional languages.

By removing these hidden information leaks, functional programming languages offer the possibility of much cleaner programs which are easier to design and debug. However, they also offer other benefits.

Many programmers accustomed to the imperative paradigm find it difficult to learn functional programming, which encompasses a whole different way of composing programs. This difficulty, along with the fact that functional programming environments do not have the extensive tools and libraries available for traditional programming languages, are among the main reasons that functional programming has received little use in the software industry. Functional languages have remained largely the domain of academics and hobbyists, and what little inroads have been made are due to impure functional languages such as Erlang and Common Lisp. It could be argued that the largest influence of functional programming on the software industry has been by those academically trained programmers who have gone on to apply the impure functional programming style to their work in traditional imperative languages.

Greater expressiveness: "new forms of glue"

A powerful mechanism sometimes used in functional programming is the notion of higher-order functions. Functions are higher-order when they can take other functions as arguments, and/or return functions as results. (The derivative in calculus is a common example of a function that maps a function to a function). Higher-order functions were studied long before the notion of functional programming existed, in the lambda calculus, a formalism which has influenced the design of several functional programming languages, especially the Haskell programming language.

Speed and space considerations

Functional languages have long been criticised as resource-hungry, both in terms of CPU resources and memory. This was mainly due to two things:
some early functional languages were implemented with no effort at efficiency

non-functional languages achieved speed at least in part by leaving out features such as bounds-checking or garbage collection which are viewed as essential parts of modern computing frameworks, the overhead of which was built-in to functional languages by default

As modern imperative languages and their implementations have started to include greater emphasis on correctness, rather than raw speed, and the implementations of functional languages have begun to emphasise speed as well as correctness, the performance of functional languages and imperative languages has begun to converge. For programs which spend most of their time doing numerical computations, functional languages can be very fast (Objective Caml and Clean often compare to C in speed). However, purely functional languages can be considerably slower when manipulating large data structures, due to less efficient memory usage.

For further reading

Cousineau, Guy and Michel Mauny. The Functional Approach to Programming. Cambridge, UK: Cambridge University Press, 1998.
Graham, Paul. ANSI Common LISP. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages." ACM Computing Surveys 21, no. 3 (1989): 359-411.
Pratt, Terrence, W. and Marvin V. Zelkowitz. Programming Languages: Design and Implementation. 3rd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
Salus, Peter H. Functional and Logic Programming Languages. Vol. 4 of Handbook of Programming Languages. Indianapolis, Indiana: Macmillan Technical Publishing, 1998.
Thompson, Simon. Haskell: The Craft of Functional Programming. Harlow, England: Addison-Wesley Longman Limited, 1996.

See also:
lazy evaluation
eager evaluation
logic programming

External links
Why Functional Programming Matters
"Functional Programming"-- Chapter 4 of Advanced Programming Language Design by Raphael Finkel, is an excellent introduction to and explanation of functional programming

Object-oriented programming

Object-oriented programming (OOP) is a computer programming paradigm that emphasizes the following aspects:

1. Objects - packaging data and functionality together into units within a running computer program; objects are the basis of modularity and structure in an object-oriented computer program.
2. Abstraction - combining multiple smaller operations into a single unit that can be referred to by name.
3. Encapsulation - separating implementation from interfaces.
4. Polymorphism - using the same name to invoke different operations on objects of different data types.
5. Inheritance - defining objects data types as extensions and/or restrictions of other object data types.

Notes: Abstraction is important to but not unique to OOP. Reusability is a benefit often attributed to OOP.

OOP is often called a paradigm rather than a style or type of programming to emphasize the point that OOP can change the way software is developed, by changing the way that programmers and software engineers think about software.

Basics of object-oriented programming

The fundamental aspect of object-oriented programming is that a computer program is composed of a collection of individual units, or objects. To make the computation happen, each object is capable of receiving messages and sending messages to other objects. In this way, messages can be handled by one chunk of code but may be done so by another code block seamlessly. It is claimed then that this gives more flexibility over just step-by-step programming, called imperative programming or structured programming in the field of computer science. Thus, the challenge OOP programmers face is how to distribute responsibility over objects, or classes--one of many popular implementation schemes.

Proponents of OOP also claim that OOP is more intuitive and, for those new to computer programming, is easier to learn than a traditional way--breaking the computation into procedures. In OOP, objects are simple, self contained and easily identifiable. This modularity allows the program parts to correspond to real aspects of the problem and thereby to model the real world. Indeed, object-oriented programming often begins from a written statement of the problem situation. Then by a process of inserting objects or variables for nouns, methods for verbs and attributes for adjectives, a good start is made on a framework for a program that models, and deals with, that situation. This allows one to learn how to program in object-oriented languages.

The majority of the computer programmers agree that OOP is a major advance on the previous complexities of procedure based methods, as its popularity attests to the fact, and OOP can be a major advantage in large projects where procedural methods tended to develop very complicated conditional loops and branches, difficult to understand and to maintain.

Implementation techniques

OOP with procedural languages
In procedural languages, OOP often appears as a form where data types are extended to behave like an object in OOP, very similar to an abstract data type with an extension such as inheritance. Each method is actually a subprogram which is syntactically bound to a class.

Definition

The definitions of OOP are disputed. In the most general sense, object-oriented programming refers to the practice of viewing software primarily in terms of the "things" (objects) it manipulates, rather than the actions it performs. Other paradigms such as functional and procedural programming focus primarily on the actions, with the objects being secondary considerations; in OOP, the situation is reversed.

Widely-used terminology distinguishes object-oriented programming from object-based. The former is held to include inheritance (described below), while the latter does not. See Dispute over the definition of object-oriented programming

Class-based models

The most popular and developed model of OOP is a class-based model, as opposed to an object-based model. In this model, objects are entities that combine both state (i.e., data) and behavior (i.e., procedures, or methods). An object is defined by a class, which is a definition, or blueprint, of all objects of a specific type. An object must be explicitly created based on a class and an object thus created is considered to be an instance of that class. An object is similar to a structure, with the addition of method pointers, member access control, and an implicit data member which locates instances of the class (i.e.: actual objects of that class) in the class hierarchy (essential for runtime inheritance features).

Inheritance

One object's data and/or functionality may be based on those of other objects, from which the former object is said to inherit. This allows commonalities among different kinds of objects to be expressed once and reused multiple times. Inheritance is also commonly held to include subtyping, whereby one type of object is defined to be a more specialised version of another type (see Liskov substitution principle), though non-subtyping inheritance is also possible. Inheritance is typically expressed by describing classes of objects arranged in an inheritance hierarchy reflecting common behavior.

Critique

Class-based languages, or, to be more precise, typed languages, where subclassing is the only way of subtyping, have been criticized for mixing up implementations and interfaces--the essential principle in object-oriented programming. This website gives a good example. It says one might create a bag class that stores a collection of objects, then extends it to make a new class called a set class where the duplication of objects is eliminated. Now, a function that takes a bag class may expect that adding two objects increases the size of a bag by two, yet if one passes an object of a set class, then adding two objects may or may not increase the size of a bag by two. The problem arises precisely because subclassing implies subtyping even in the instances where the principle of subtyping, known as the Liskov Substitution Principle, does not hold.

Also, another common example is that a person object created from a child class cannot become an object of parent class because a child class and a parent class inherit a person class but class-based languages mostly do not allow to change the kind of class of the object at runtime.

Prototype-based models

Other than using classes, prototyping is another, less popular, means of achieving object-oriented behavior sharing. After an object is defined, another similar object will be defined by referring to the original one as a template, then listing the new object's differences from the original. SELF, a programming language developed by Sun Microsystems is an instance of a language that uses prototyping for behavior sharing rather than classification. NewtonScript, Act1 and DELEGATION are other examples. Hybrid and Exemplars use both prototyping and classification. In prototyping systems, objects themselves are the templates, while classification systems use classes as templates for objects.

The classification approach is so predominant in OOP that many people would define objects as encapsulations that share data by classification and inheritance. However, the more generic term "behavior sharing" acknowledges alternate techniques such as prototyping.

Object-based models

Object-based programming techniques include the concept of an object (encapsulation, and abstraction) but do not include the class-based models of inheritance.

History

The concept of objects and instances in computing had its first major breakthrough with Sketchpad made by Ivan Sutherland in 1963. However this was an application and not a programming paradigm. The object-oriented programming paradigm first took root in Simula 67, a language designed for making simulations, created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Centre in Oslo. (Reportedly, the story is that they were working on ship simulations, and were confounded by the combinatorial explosion of how the different attributes from different ships could affect one another. The idea occurred to group the different types of ships into different classes of objects, each class of objects being responsible for defining its own data and behavior.) They were later refined in Smalltalk, which was developed in Simula at Xerox PARC, but was designed to be a fully dynamic system in which objects could be created and modified "on the fly" rather than having a system based on static programs.

Object-oriented programming "took off" as the dominant programming methodology during the mid-1980s, largely due to the influence of C++, an extension of the C programming language. Its dominance was further cemented by the rising popularity of Graphical user interfaces, for which object-oriented programming is allegedly well-suited. Indeed, the rise of GUIs changed the user focus from the sequential instructions of text-based interfaces to the more dynamic manipulation of tangible components. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective C, an object-oriented, dynamic messaging extension to C based on Smalltalk.

At ETH Zurich, Niklaus Wirth and his colleagues had also been investigating such topics as data abstraction and modular programming. Modula-2 included both, and their succeeding

design, Oberon included a distinctive approach to object orientation, classes, and such. The approach is unlike Smalltalk, and very unlike C++.

Object-oriented features have been added to many existing languages during that time, including Ada, BASIC, Lisp, Pascal, and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code. "Pure" object-oriented languages, on the other hand, lacked features that many programmers had come to depend upon. To bridge this gap, many attempts have been made to create new languages based on object-oriented methods but allowing some procedural features in "safe" ways. Bertrand Meyer's Eiffel was an early and moderately successful language with those goals.

In the past decade Java has emerged in wide use partially because of its similarity to C language and to C++, but perhaps more importantly because if its implementation using a virtual machine that is intended to run code unchanged on many different platforms. This last feature has made it very attractive to larger development shops with heterogeneous environments. Microsoft's .NET initiative has a similar objective and includes/supports several new languages, or variants of older ones.

More recently, a number of languages have emerged that are primarily object-oriented yet compatible with procedural methodology, such as Python and Ruby. Besides Java, probably the most commercially important recent object-oriented languages are VB.NET and C Sharp designed for Microsoft's .NET platform.

Just as procedural programming led to refinements of techniques such as structured programming, modern object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modelling languages (such as UML).

Further reading

Grady Booch: Object-Oriented Analysis and Design with Applications, Addison-Wesley, ISBN 0805353402
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, ISBN 0201633612
Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall, ISBN 0136291554
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: Object-Oriented Modeling and Design, Prentice Hall, ISBN 0136298419
Ivar Jacobsen: Object-Oriented Software Engineering: A Use Case-Driven Approach, Addison-Wesley, ISBN 0201544350
Harold Abelson, Gerald Jay Sussman, Julie Sussman: Structure and Interpretation of Computer Programs, The MIT Press, ISBN 0262011530
Paul Harmon, William Morrissey: The Object Technology Casebook - Lessons from Award-Winning Business Applications, John Wiley & Sons, ISBN 0-471-14717-6
David A. Taylor: Object-Oriented Information Systems - Planning and Implementation, John Wiley & Sons, ISBN 0-471-54364-0
Peter Eeles, Oliver Sims: Building Business Objects, John Wiley & Sons, ISBN 0-471-19176-0

See also:

Software component
Interface description language

object-based programming
Object-oriented programming language
Functional programming
Procedural programming
Structured programming
Post-object programming
Aspect-oriented programming
Subject-oriented programming
Intentional programming
Reflective programming
Distributed programming
glossary of object-oriented programming
How to program in object-oriented languages
Design pattern
Refactoring
CORBA
Globus
DCOM

External links
Object-oriented programming FAQ
Criticism of OOP

Referential transparency

In computer programming, a referentially transparent function is one that, given the same parameter(s), it always returns the same result.

While in mathematics all functions are referentially transparent, in programming this is not always the case. For example, take a "function" that takes no parameters and returns input from the keyboard. A call to this function may be GetInput(). The return value of GetInput() depends on what the user feels like typing in, so multiple calls to GetInput() with identical parameters (the empty list) may return different results.

A more subtle example is that of a "function" that uses a global variable to help it compute its results. Since this variable isn't passed as a parameter but can be altered, the results of subsequent calls to the function can differ even if the parameters are identical.

Why is referential transparency important? Because it allows the programmer to reason about program behavior, which can help in proving correctness, finding bugs that couldn't be found by testing, simplifying the algorithm, assisting in modifying the code without breaking it, or even finding ways of optimizing it.

Some functional programming languages enforce referential transparency for all functions.

Function object (functor)

In functional programming languages such as ML, a function object or functor represents a mapping that maps modules to modules, representing a tool for the reuse of code, and is used in a manner analogous to the original mathematical meaning of the term.

Recently, in C++ and Java, the term functor was coined, probably independently, with a meaning largely unrelated to category theory. In object-oriented software design a class can contain both data and function. The term functor was chosen to describe objects that represent the application of a function to (dynamic) parameter data.

This sort of functor has some of the characteristics that a function pointer would have in procedural languages such as C. In a procedural program, a sort function might accept a pointer to a "plain" function defining the ordering relation between items to be sorted. In a more object-oriented style, a sort method would accept a functor object that exposes a method defining the ordering.

In Java, the sort method signature would define the functor parameter in terms of a base class (typically an interface). In C++ the sort method would expect the functor class to define an appropriate application operator (). C++ also allows the possibility that the functor class itself is specified through a template argument.

Functors are more powerful than function pointers in that they may contain state (data) that can be used by the function represented. Compilers are sometimes able to generate improved code through inlining.

A functor often contains a single public method apart from the constructor, although this is not a strict limitation.

Lambda calculus

The lambda calculus is a formal system designed to investigate function definition, function application and recursion. It was introduced by Alonzo Church and Stephen Kleene in the 1930s; Church used the lambda calculus in 1936 to give a negative answer to the Entscheidungsproblem. The calculus can be used to cleanly define what a "computable function" is. The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm, and this was the first question, even before the halting problem, for which undecidability could be proved. Lambda calculus has greatly influenced functional programming languages, especially Lisp.

This article deals with the "untyped lambda calculus" as originally conceived by Church. Since then, some typed lambda calculi have been developed.

History

Originally, Church had tried to construct a complete formal system for the foundations of mathematics; when the system turned out to be susceptible to the analog of Russell's paradox, he separated out the lambda calculus and used it to study computability, culminating in his negative answer to the Entscheidungsproblem.

Informal description

In lambda calculus, every expression stands for a function with a single argument; the argument of the function is in turn a function with a single argument, and the value of the function is another function with a single argument. Functions are anonymously defined by a lambda expression which expresses the function's action on its argument. For instance, the "add-two" function f(x) = x + 2 would be expressed in lambda calculus as λ x. x + 2 (or equivalently as λ y. y + 2; the name of the formal argument is immaterial) and the number f(3) would be written as (λ x. x + 2) 3. Function application is left associative: f x y = (f x) y. Consider the function which takes a function as argument and applies it to the argument 3: λ x. x 3. This latter function could be applied to our earlier "add-2" function as follows: (λ x. x 3) (λ x. x+2). It is clear that the three expressions
(λ x. x 3) (λ x. x+2) and (λ x. x + 2) 3 and 3 + 2
are equivalent. A function of two variables is expressed in lambda calculus as a function of one argument which returns a function of one argument (see Currying). For instance, the function f(x, y) = x - y would be written as λ x. λ y. x - y. The three expressions
(λ x. λ y. x - y) 7 2 and (λ y. 7 - y) 2 and 7 - 2
are equivalent. It is this equivalence of lambda expressions which in general can not be decided by an algorithm.

Not every lambda expression can be reduced to a definite value like the ones above; consider for instance
(λ x. x x) (λ x. x x)
or
(λ x. x x x) (λ x. x x x)
and try to visualize what happens as you start to apply the first function to its argument. ((λ x. x x) is also known as the ω combinator; ((λ x. x x) (λ x. x x)) is known as Ω, (λ x. x x x) (λ x. x x x) as Ω2, etc.)

While the lambda calculus itself does not contain symbols for integers or addition, these can be defined as abbreviations within the calculus and arithmetic can be expressed as we will see below.

Lambda calculus expressions may contain free variables, i.e. variables not bound by any λ. For example, the variable y is free in the expression (λ x. y), representing a function which always produces the result y. Occasionally, this necessitates the renaming of formal arguments, for instance in order to reduce
(λ x. λ y. y x) (λ x. y) to λ z. z (λ x. y)

If one only formalizes the notion of function application and does not allow lambda expressions, one obtains combinatory logic.

Formal definition

Formally, we start with a countably infinite set of identifiers, say {a, b, c, ..., x, y, z, x1, x2, ...}. The set of all lambda expressions can then be described by the following context-free grammar in BNF:
<expr> → <identifier>
<expr> → (λ <identifier> . <expr>)
<expr> → (<expr> <expr>)

The first two rules generate functions, while the third describes the application of a function to an argument. Usually the brackets for lambda abstraction (rule 2) and function application (rule 3) are omitted if there is no ambiguity under the assumptions that (1) function application is left-associative, and (2) a lambda binds to the entire expression following it. For example, the expression ((λ x. (x x)) (λ y. y)) can be simply written as (λ x. x x) λ y.y.

Lambda expressions such as λ x. (x y) do not define a function because the occurrence of the variable y is free, i.e., it is not bound by any λ in the expression. The binding of occurrences of variables is (with induction upon the structure of the lambda expression) defined by the following rules:
In an expression of the form V where V is a variable this V is the single free occurrence.
In an expression of the form λ V. E the free occurrences are the free occurrences in E except those of V. In this case the occurrences of V in E are said to be bound by the λ before V.
In an expresssion of the form (E E') the free occurrences are the free occurrences in E and E' .

Over the set of lambda expressions an equivalence relation (here denoted as ==) is defined that captures the intuition that two expressions denote the same function. This equivalence relation is defined by the so-called alpha-conversion rule and the beta-reduction rule.

α-conversion

The alpha-conversion rule is intended to express the idea that the names of the bound variables are unimportant; for example that λx.x and λy.y are the same function. However, the rule is not as simple as it first appears. There are a number of restrictions on when one bound variable may be replaced with another.

The alpha-conversion rule states that if V and W are variables, E is a lambda expression and E[V/W] means the expression E with every free occurrence of V in E replaced with W then
λ V. E == λ W. E[V/W]

if W does not appear freely in E and W is not bound by a λ in E whenever it replaces a V. This rule tells us for example that λ x. (λ x. x) x is the same as λ y. (λ x. x) y.

β-reduction

The beta-reduction rule expresses the idea of function application. It states that
((λ V. E) E') == E[V/E']
if all free occurrences in E' remain free in E[V/E'].

The relation == is then defined as the smallest equivalence relation that satisfies these two rules.

A more operational definition of the equivalence relation can be given by applying the rules only from left to right. A lambda expression which does not allow any beta reduction, i.e., has no subexpression of the form ((λ V. E) E'), is called a normal form. Not every lambda expression is equivalent to a normal form, but if it is, then the normal form is unique up to naming of the formal arguments. Furthermore, there is an algorithm for computing normal forms: keep replacing the first (left-most) formal argument with its corresponding concrete argument, until no further reduction is possible. This algorithm halts if and only if the lambda expression has a normal form. The Church-Rosser theorem then states that two expressions result in the same normal form up to renaming of the formal arguments if and only if they are equivalent.

η-conversion

There is third rule, eta-conversion, which may be added to these two to form a new equivalence relation. Eta-conversion expresses the idea of extensionality, which in this context is that two functions are the same iff they give the same result for all arguments. Eta-conversion converts between λ x . f x and f, whenever x does not appear free in f. This can be seen to be equivalent to extensionality as follows:

If f and g are extensionally equivalent, i.e. if f a == g a for all lambda expressions a, then in particular by taking a to be a variable x not appearing free in f we have f x == g x and hence λ x . f x == λ x . g x, and so by eta-conversion f == g. So if we take eta-conversion to be valid, we find extensionality is valid.

Conversely if extensionality is taken to be valid, then since by beta-reduction for all y we have (λ x . f x) y == f y, we have λ x . f x == f - i.e. eta-conversion is found to be valid.

Arithmetic in lambda calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church integers, which can be defined as follows:
0 = λ f. λ x. x
1 = λ f. λ x. f x
2 = λ f. λ x. f (f x)
3 = λ f. λ x. f (f (f x))
and so on. Intuitively, the number n in lambda calculus is a function that takes a function f as argument and returns the n-th power of f. (Note that in Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of 0 impossible.) Given this definition of the Church integers, we can define a successor function, which takes a number n and returns n + 1:
SUCC = λ n. λ f. λ x. f (n f x)

Addition is defined as follows:
PLUS = λ m. λ n. λ f. λ x. m f (n f x)
PLUS can be thought of as a function taking two natural numbers as arguments and returning a natural number; it is fun to verify that
PLUS 2 3 and 5
are equivalent lambda expressions. Multiplication can then be defined as
MULT = λ m. λ n. m (PLUS n) 0,
the idea being that multiplying m and n is the same as m times adding n to zero. Alternatively
MULT = λ m. λ n. λ f. m (n f)
The predecessesor PRED n = n - 1 of a positive integer n is more difficult:
PRED = λ n. λ f. λ x. n (λ g. λ h. h (g f)) (λ u. x) (λ u. u)
or alternatively
PRED = λ n. n (λ g. λ k. (g 1) (λ u. PLUS (g k) 1) k) (λ l. 0) 0
Note the trick (g 1) (λ u. PLUS (g k) 1) k which evaluates to k if g(1) is zero and to g(k) + 1 otherwise.

Logic and predicates

By convention, the following two definitions are used for the boolean values TRUE and FALSE:
TRUE = λ u. λ v. u
FALSE = λ u. λ v. v
A predicate is a function which returns a boolean value. The most fundamental predicate is ISZERO which returns true if and only if its argument is zero:
ISZERO = λ n. n (λ x. FALSE) TRUE
The availability of predicates and the above definition of TRUE and FALSE make it convenient to write "if-then-else" statements in lambda calculus.

Recursion

Recursion is the definition of a function using the function itself; on the face of it, lambda calculus does not allow this. However, this impression is misleading. Consider for instance the factorial function f(n) recursively defined by

f(n) = 1, if n = 0; and n·f(n-1), if n>0.

One may view the right-hand side of this definition as a function g which takes a function f as an argument and returns another function g(f). Using the ISZERO predicate, the function g can be defined in lambda calculus. The factorial function is then a fixed-point of g:
f = g(f).
In fact, every recursively defined function can be seen as a fixed point of some other suitable function. This allows the definition of recursive functions in lambda calculus, because every function in lambda calculus has a fixed point, and the fixed point can be easily described: the fixed point of a function g is given by
(λ x. g (x x)) (λ x. g (x x))
and if we define the Y combinator as
Y = λ g. (λ x. g (x x)) (λ x. g (x x))
then Y g is a fixed point of g, meaning that the two expressions
Y g and g (Y g)
are equivalent. Using Y, every recursively defined function can be expressed as a lambda expression. In particular, we can now cleanly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

Computability and lambda calculus

A function F : N → N of natural numbers is defined to be computable if there exists a lambda expression f such that for every pair of x, y in N, F(x) = y if and only if the expressions f x and y are equivalent. This is one of the many ways to define computability; see the Church-Turing thesis for a discussion of other approaches and their equivalence.

Undecidability of equivalence

There is no algorithm which takes as input two lambda expressions and output "YES" or "NO" depending on whether or not the two expressions are equivalent. This was historically the first problem for which the unsolvability could be proven. Of course, in order to do so, the notion of "algorithm" has to be cleanly defined; Church used a definition via recursive functions, which is now known to be equivalent to all other reasonable definitions of the notion.

Church's proof first reduces the problem to determining whether a given lambda expression has a normal form. A normal form is an equivalent expression which cannot be reduced any further. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Building on earlier work by Kleene and utilizing Gödel's procedure of Gödel numbers for lambda expressions, he constructs a lambda expression e which closely follows the proof of Gödel's first incompleteness theorem. If e is applied to its own Gödel number, a contradiction results.

Lambda calculus and programming languages

Most functional programming languages are equivalent to lambda calculus extended with constants and datatypes. Lisp uses a variant of lambda notation for defining functions but only its purely functional subset is really equivalent to lambda calculus.

References

Stephen Kleene, A theory of positive integers in formal logic, American Journal of Mathematics, 57 (1935), pp 153 - 173 and 219 - 244. Contains the lambda calculus definitions of several familiar functions.
Alonzo Church, An unsolvable problem of elementary number theory, American Journal of Mathematics, 58 (1936), pp 345 - 363. This paper contains the proof that the equivalence of lambda expressions is in general not decidable.
Jim Larson, An Introduction to Lambda Calculus and Scheme. A gentle introduction for programmers.
Martin Henz, The Lambda Calculus. Formally correct development of the Lambda calculus.
Henk Barendregt, The lambda calculus, its syntax and semantics, North-Holland (1984), is the comprehensive reference on the (untyped) lambda calculus.
Amit Gupta and Ashutosh Agte, Untyped lambda-calculus, alpha-, beta- and eta- reductions and recursion

External links
L. Allison, Some executable λ-calculus examples
Georg P. Loczewski, The Lambda Calculus and A++

Currying

In computer science, currying is a style of treating multiple-argument functions as single-argument functions, named after the logician Haskell Curry, though it was invented by Moses Schönfinkel. A curried function typically consumes the first argument evaluating to another function, which consumes the second argument, evaluating to ... and so on, until the last argument is consumed, evaluating to the result. For example, using lambda calculus notation, a curried sum function is λ x . λ y . x + y; the λ-expression (λ x . λ y . x + y) 3 4 is first β-reduced to (λ y . 3 + y) 4 and then β-reduced to 3 + 4, which is then δ-reduced to 7. Currying is popular in functional programming; it is contrasted by the non-currying style favoured by mathematicians, where multiple arguments are converted to a single argument by packing the arguments into a tuple that forms the single argument.

In functional programming languages, currying is an operation performed on functions of more than one argument. Currying a function f of two arguments produces a function g of one argument that returns a function of one argument such that f(x, y) equals (g(x))(y), or in Lisp notation (f x y) equals ((g x) y). By extension, fully currying a function f of three arguments produces g such that f(x, y, z) equals ((g(x))(y))(z), or in Lisp notation (f x y z) equals (((g x) y) z).

To do currying in the Scheme programming language:

```
(define curry2
  (lambda (f)
    (lambda (x)    ; take the first argument
      (lambda y    ; and the rest of the args as a list
        (f x . y)))))
```

If g equals (curry2 f), then (f x y) equals ((g x) y), and (f x y z) equals ((g x) y z).

These languages automatically fully curry functions called with too few arguments:
ML
Haskell

See also:
lazy evaluation

Design patterns

Design patterns are standard solutions to common problems in object-oriented software design. The phrase was introduced to computer science in 1995 by the text Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0201633612). The scope of the term remained a matter of dispute into the next decade. Algorithms are not thought of as design patterns, since they solve implementation problems rather than design problems. Typically, a design pattern is thought to encompass a tight interaction of a few classes and objects.

History

Christopher Alexander was the first to mention the idea of patterns. He is an architect and an urban planner. Alexander thought about architecture and planning as a continuous reapplication of simple principles. He decided to extract these principles in a form independent from the specific problems they are used to solve. He started out by using mathematical methods to develop these principles, and then he defined a pattern language to reduce the complexity of formal methods. This idea is explained in his thesis Notes on the Synthesis of Form. Alexander wrote other books which included enhancements of his idea. In the software field, patterns were first devised in the late 1980's. At that time, Ward Cunningham and Kent Beck were working on a project that involved designing user interface through Smalltalk. Due to difficulties and lack of good methods for designing user interface, they decided to use the idea of patterns that Christopher introduced. They designed five patterns for Smalltalk windows. Later on, they started writing a complete pattern language. They introduced ten other patterns and aimed to develop up to 150 patterns. They explained their work in a paper titled: Using Pattern Languages for Object-Oriented Programs. In 1991, Jim Coplien was developing a catalog about idioms. Idioms are patterns that are specific to a particular programming language, in this case C++. Coplien published his work in a book titled Advanced C++: Programming Styles and Idioms. At around the same time, Erich Gamma was working on his PhD thesis about object oriented software development. He realized the benefit of recording frequent design structures. Coplien and Gamma and others met at a series of OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) workshop and exchanged their ideas about patterns.

Four people (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – known collectively as the Gang of Four) wrote the first book about design patterns entitled Design Patterns – Elements of Reusable Object-Oriented Software. In this book, they classified patterns into Creational, Behavioral, and Structural patterns. They introduced around 22 patterns. They presented a way to document design patterns. This book is considered as the basic reference to design patterns and many works were based on it. In 1993, a small group of

people interested in patterns formed the Hillside Generative Patterns Group. They formed the first conference on patterns Pattern Languages of Programming (PLoP) in 1994.

Advantages

A very important advantage of design patterns is the fact that they speed up the development process by providing an almost ready made solution that has been used earlier and proved to be efficient. Commonly used design patterns also have the potential of being revised and improved over time, and thus are more likely to perform better than home made designs.

Moreover, design patterns allow for a generalized solution that does not depend on understanding a specific design problem from all its aspects, and thus ease reusing this solution. There is a possibility that someone had experienced your problem in a similar context and found a suitable solution for it. However, as it is, you need to understand the details of this person's design in order to adopt the same solution to your problem. If this person documented his or her solution in the format of a design pattern, you would not have to grasp these details. Additionally, having documented a design pattern will make it easier to recognize a problem in one's design and thus find a solution directly. Moreover, the fact that this design pattern is documented with a specific name makes it easier to communicate about it among developers.

Classification

Design patterns can be classified based on multiple criteria. The most common of which is their classification based on the problem they solve. According to this criterion, design patterns can be classified various classes, some of which are:

Fundamental patterns
Creational Patterns
Structural patterns
Behavioral patterns
Concurrency patterns
Real-time patterns

Documentation

The documentation for a design pattern should contain enough information about the problem the pattern addresses, the context in which it is used, and the suggested solution. Nonetheless, authors use their own layouts to document design patterns, and these layouts usually resemble the essential parts. The authors usually include additional sections to provide more information, and organize the essential parts in different sections, possibly with different names. A commonly used format is the one used by the Gang of Four. It contains the following sections:

Pattern Name and Classification: Every pattern should have a descriptive and unique name that helps in identifying and referring to it. Additionally, the pattern should be classified according to a classification such as the one described earlier. This classification helps in identifying the use of the pattern.

Intent: This section should describe the goal behind the pattern and the reason for using it. It resembles the problem part of the pattern.
Also Known As: A pattern could have more than one name. These names should be documented in this section.
Motivation: This section provides a scenario consisting of a problem and a context in which this pattern can be used. By relating the problem and the context, this section shows when this pattern is used.
Applicability: This section includes situations in which this pattern is usable. It represents the context part of the pattern.
Structure: A graphical representation of the pattern. Class diagrams and Interaction diagrams can be used for this purpose.
Participants: A listing of the classes and objects used in this pattern and their roles in the design.
Collaboration: Describes how classes and objects used in the pattern interact with each other.
Consequences: This section describes the results, side effects, and trade offs caused by using this pattern.
Implementation: This section describes the implementation of the pattern, and represents the solution part of the pattern. It provides the techniques used in implementing this pattern, and suggests ways for this implementation.
Sample Code: An illustration of how this pattern can be used in a programming language
Known Uses: This section includes examples of real usages of this pattern.
Related Patterns: This section includes other patterns that have some relation with this pattern, so that they can be used along with this pattern, or instead of this pattern. It also includes the differences this pattern has with similar patterns.

Critique

Some feel that the need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. It is also said that design patterns encourage navigational database-like structures instead of the allegedly cleaner relational approach where such structures are viewpoints instead of hard-wired into programming code. However, critics of the relational approach suggest that it does not integrate well enough with behavior. The level of coupling that should be supplied between behavior and data is a contentious topic.

Related topics

Pattern mining
Programming practice
Refactoring
Software engineering and List of software engineering topics

External links

One of many Perl design patterns wiki sites
Many design patterns are described in mgrand's book
Patterns Catalog
The Object-Oriented PatternDigest
Design Pattern Toolkit
XML structural design patterns.
Perl Design Patterns

Container

In computer programming, container pattern is one of design patterns in which objects are created to hold other objects. Queues, FIFOs/stacks, buffers, shopping carts, and caches all fit this description.

Breadth first recursion has an example of recursing through a network of objects to find them all, where a queue is used to hold unexplored paths.

Iterator interface is an important part of all objects that act as containers in one way or another. It provides a consistent way to loop through that containers contents: any container should be functionally interchangeable with any other for the purposes of inspecting their contents. This employes the ideas of abstract root classes and abstract class.

Template class talks about generators for containers. Type safety breaks down when presented with generic, reusable containers that can hold any type of data. If a container only holds one specific type of data, we know any items retrieved from it are of the correct type, and no type errors can occur, but then we can't reuse that container. TemplateClass follows C++'s ideas of templates, and provides a generic implementation that can create instances tailored to specific data types to enforce safety. ObjectOriented purists will find this of interest.

Aggregation pattern and state vs class talk about other, more present, type issues that crop up when creating containers full of subclasses of a certain type. What if one subclass doesn't do something the superclass does? Model it as state. Null-methods are okay. Don't fork the inheritance to remove a feature. Similar to Introduce null object, but for methods.

Object oriented design heuristics, section 5.19, has an example of a basket that cores fruit. How could this possibly made general? Anything other than a fruit would need a //->core()// method that does nothing, requiring a base class implementing a stub //core()// to be inherited by all.

Extract a generic interface:
Generalize - Rather than //core()//, why not //prepare()//? Oranges

could peel themselves, grapes devine themselves, and so forth. Method calls aren't instructions on how to do something but rather a request that an end be achieved. How it is done is best left to the object.
Extract interface - Given a saner interface, make it optional.

Let the basket test //->can('prepare')//. If the item is capable of doing so, it may. If it isn't, no big deal. The magic basket prepares fruit. Not preparing non-fruit is okey. No one ever said just because it prepares fruit it has to blow up when presented with non-fruit. This is somewhat of a compromise - TypeSafety doesn't exist for things wishing to use the basket as a repository for all things fruit and no thing not fruit. Useful for avoiding interfacebloat

The article is originally from Perl Design Patterns Book

Iterator

In computer programming, an iterator is an object that maintains a type of cursor used for processing each element in a list or in another data structure that contains similar items. Iterators are frequently used in languages like Java; they are crucial in gaming languages such as UnrealScript. An example usage of an iterator (for java) can be explained here.

```
Iterator it = list.iterator();
while(it.hasNext()){
  Object spot = it.next();
}
```

The above example implies that the Object (called list) supports an iterator method. This hypothetical iterator then supplies the methods hasNext and next that are used for scrolling through the list of items.

The variable spot is a type of cursor marking the current element.

```
foreach $spot (@list){
   print "$spot\n";
}
```

The above example (in perl) shows the same logic, but is not object oriented. Here @list is an Array of items. The iterator is the foreach keyword, and is used to traverse the list.

Each element in the list is represented by the variable $spot the code within the block can then operate on that particular element.

Physically, an iterator can be viewed as a ruler or (other marker, such as a human finger) placed over a paper list. Each entry is "pointed at" by the ruler, to get to the next entry, one would slide the ruler down one line. This implements a type of physical iterator.

One might ask why have iterators when most languages support counting loops?
Counting loops are not suited to all data structures, in particular to data structures with no or slow random access, like lists.
Iterators often provide a consistent way to iterate on data structures of all kinds, and so their use help make the code more portable, reusable, less sensitive to a change of data structure. (They are heavily used for that purpose by the C++ standard template library).
It is common for the number of items in a list to change even while it is being iterated. An iterator can usually be set up to deal with this without the programmer having to take into account all of the various changes that could occur. This has become very necessary in modern object oriented programming, where the interrelationships between objects and the effects of operations may not be obvious - by using an iterator one is isolated from these sorts of consequences.

See Also: Iterator pattern, Iteration

Abstract data type

Abstract data types or ADTs are data types described in terms of the operations they support—their interface—rather than how they are implemented.

For example, a sequence (or list) could be defined as follows. A sequence contains a variable number of ordered elements, and supports the following operations:
create a new, empty sequence
get a handle to the first element of a sequence
get a handle to the next element of a sequence
get a handle to the last element of a sequence
get a handle to the previous element of a sequence
get a handle to an arbitrary element of a sequence
insert an element at the beginning of a sequence
insert an element at the end of a sequence
insert an element after an element of a specific handle
delete the first element of a sequence
delete the last element of a sequence
delete the element at a specific handle
delete all elements between two handles

Where a handle is associated with a single element and allows the following operations:
get the value of the associated element of this handle
modify the value of the associated element of this handle

Arrays, linked lists, and binary trees—among other data structures—can all support these operations, with different performance tradeoffs. Arrays are fast at accessing the previous, next, first, last or arbitrary element, but slow at inserting or deleting items from anywhere but the very end; singly-linked lists are fast at accessing the first or next element, as well as adding or deleting items after a given handle, but slow at accessing arbitrary, the previous, or the last element; and binary trees are relatively fast at all the above operations, but not as fast as arrays or linked lists are at the specific operations for which they are each fastest.

Some programming languages, such as Ada and Modula-2, have explicit support for abstract data types. Object-oriented languages carry this a step further by adding inheritance and polymorphism to ADTs to get "objects".

Multiple dispatch

A feature of certain systems for object-oriented programming, such as the Common Lisp Object System, whereby a method can be specialized on more than one of its arguments. In "conventional" object-oriented programming languages, when you invoke a method ("send a message" in Smalltalk, "call a member function" in C++) one of its arguments is treated specially and used to determine which of the (potentially many) methods of that name is to be applied. In languages with multiple dispatch, all the arguments are available for the run-time selection of which method to call. This may be made clearer by an example. Imagine a game which has, among its (user-visible) objects, spaceships and asteroids. When two objects collide, the program may need to do different things according to what has just hit what. In a language with only single dispatch, such as C++, the code would probably end up looking something like this:

```
void Asteroid::collide_with(Thing * other) {
  Asteroid * other_asteroid = dynamic_cast<Asteroid>(other);
  if (other_asteroid) {
    // deal with asteroid hitting asteroid
    return;
  }
  Spaceship * other_spaceship = dynamic_cast<Spaceship>(other);
  if (other_spaceship) {
    // deal with asteroid hitting spaceship
    return;
  }
}
```

with one collide_with member function for each kind of object that can collide with others, each containing one case per class. In a language with multiple dispatch, such as Common Lisp, it might look more like this:

```
(defmethod collide-with ((x Asteroid) (y Asteroid))
  ;; deal with asteroid hitting asteroid
)

(defmethod collide-with ((x Asteroid) (y Spaceship))
  ;; deal with asteroid hitting spaceship
)
```

and similarly for the other methods. No explicit testing or "dynamic casting" in sight.

In the presence of multiple dispatch, the traditional idea of methods as being defined in classes and contained in objects becomes less appealing -- each collide-with method there is attached to two different classes, not one. Hence, the special syntax for method invocation generally disappears, so that method invocation looks exactly like ordinary function invocation, and methods are grouped not in classes but in generic functions. Multiple dispatch differs from overloading in C++ in that it takes place at run time, on the dynamic types of the arguments, rather at than compile time and on the static types of the arguments.

Higher-order function

In mathematics and computer science, higher-order functions are functions which can take other functions as arguments, and may also return functions as results. The derivative in calculus is a common example of a higher-order function, since it maps a function to another function. Higher-order functions were studied long before the notion of functional programming existed, in the lambda calculus, a formalism which has influenced the design of several functional programming languages, especially the Haskell programming language.

Higher order functions in Haskell implement tremendous power in very few lines of code. For example, the following Haskell functions will square each number in a given list

```
-- without higher order functions
 squareListNoHof []   = []
 squareListNoHof list = ((head list)^2):(squareListNoHof (tail list))

 -- with higher order functions
 squareList list = map (^2) list
```

In the above example, map takes in the function (^2) (note that the first argument to (^2) is omitted, this instructs Haskell to substitute elements of the list as the first argument to the function), and the list list, and thus squares each element. map generalises the idea of "mapping" a function onto a list, that is, applying a function on to each element of a list.

The above was an example of a higher-order function that takes in a function as an argument, but does not return a function of sorts as an output. However there are standard higher order functions that do, such as the (.) function. For example, the following function will calculate the numerical equivalent to the function :

```
 -- without higher order functions
 doFunctionNoHof x = cos (log (sqrt (3x+2))

 -- with higher order functions
 doFunction x = (cos.log.sqrt) (3x+2)
```

In the above example, the (.) function takes in two functions as an argument and returns a function representing their composition: eg (f.g) x = f(g(x)). Strictly, in the above example, (cos.log.sqrt) (3x+2) is basically equivalent to (cos.(log.sqrt)), but in computer-parsing, the first expression is converted, so the notational simplification is still held.

See also: functional analysis, combinatory logic

Reflection

In computer science reflection is the ability of a program to examine and possibly modify its high level structure at runtime. It is most common in just in time compiled languages.

When program source code is compiled information about the structure of the program is normally lost as lower level code (typically assembly code) is emitted. If a system supports reflection the structure is preserved as metadata with the emitted code.

Known platforms supporting reflection are:

The Java Virtual Machine (JVM)
.NET and the Common Language Runtime (CLR)

Name binding

In computer science, binding is associating objects and implementations with names in programming language so that those objects and implementaions can be accessed by the names. An object's names are said to be "bound" to them. Deep binding and shallow bindings are not kinds of binding, but ways to implement binding. The simplest example is defining subprograms:

```
def sum (x, y)
  x + y
end
sum (2, 3)
```

In this Ruby programming language code, an implementation returning the sum of given two inputs is bound to a name, sum.

Binding time

Because objects in computer programs are usually resident in the computer memory, binding time is almost the same as the time of allocation of memory space for objects. Bind can be done either at compile-time or link-time (static binding) or at run-time (dynamic binding). Also, scope rules might define binding time of objects. Local variables, for example, are usually bound at run-time while global variables at compile-time. For example,

```
static int n;
int main ()
{
  n = 12;
  return n;
}
```

In this C code, a global variable n is bound to certain location in the memory of the computer.

Dynamic binding for polymorphism

In object-oriented programming, an object can respond to the same message with a different implementation. Dynamic binding is a common solution for this problem. For example there may be an object that contains the name (data) 'Socrates', and which is of the class (or type) Person. Now suppose all Persons are mortal. In object oriented programming, we can say that the Person class must implement the Mortal interface, which contains the method die(). Persons and Plants die in different ways, for example Plants don't stop breathing. 'Dynamic binding' is the practice of figuring out which method to invoke at runtime. For example, if we write

```
 void kill(Mortal m) {
   m.die();
 }
```

it's not clear whether m is a Person or a Plant, and thus whether Plant.die() or Person.die() should be invoked on the object. With dynamic binding, the m object is examined at runtime, and the method corresponding to it's actual class is invoked. (This implies that the actual representation of an object in memory is just its data and doesn't include the methods.)

Lazy evaluation

In computer programming, Lazy Evaluation is a concept that attempts to minimize the work the computer has to do. It has two related, yet different meanings, that could be described as delayed evaluation and minimal evaluation.

The opposite of lazy evaluation is eager evaluation, also known as strict evaluation which is the normal (and often only) evaluation method in most programming languages.

Minimal evaluation
Minimal evaluation is when an expression is only evaluated until the point where its final value is known. This means, that sometimes,not all the parts of an expression are evaluated. In the following example (in C syntax)

```
int a = 0;
if (a && myfunc(b)) {
   do_something();
}
```

minimal evalution would mean that myfunc(b) is never called. This is because a evaluates to zero, and 0 and X is 0 for any value of X. When using minimal evaluation it is important to know the expression evaluation order, which is guaranteed in some programming languages (e. g. C: left to right; Java: left to right), but not in others.

Java for example guarantees lazy evaluation and left-to-right evaluation order. Hence this example is a good code:

```
String a;
/* do something with a here */
if (a != null && a.length() > 0) {
    // do more
}
```

The C programming language guarantees left-to-right evaluation order, but not lazy evaluation. This is a compiler feature. This example is not good code, as the result depends on compiler and compiler options you have chosen:

```
char *a = NULL;
if (a != null && strlen(a) > 0) {
   do_something();
}
```

If you set your compilier to "strict evaluation" mode, you may get a segmentation fault.

Key advantages of minimal evaluation are, notably, that it allows programmers to produce infinite sequences without pushing the evaluator into an endless loop, and other neat things.

Delayed evaluation

Delayed evaluation is used in particularly in functional languages. An expression, when using delayed evaluation, is not evaluated as soon as it gets bound to a variable, but when something forces the evaluator to produce the expression's value.

In the Scheme programming language, delayed evaluation can be forced by using (define delayed-expression (delay expression)). Then (force delayed-expression) will yield the value of expression.

There are some programming languages where expressions are delay evaluated by default. Haskell is one such language.

Delayed evaluation has the added advantage of being able to create infinite lists without infinite loops or size matters in computation. One could create a function that creates an infinite list of, say, Fibonacci numbers, and the calculation of the n-th Fibonacci number would be merely the extraction of the element from that infinite list. The entire infinite list is never calculated, but only the values that influence a calculation.

Lazy evaluation as a design pattern

As well as the formal concept of lazy evaluation in programming languages, lazy evaluation is a design pattern often seen in general computer programming.

For example, in modern computer window managers, the painting of information to the screen is driven by "expose events" which drive the display code at the last possible moment. By doing this, they avoid the over-eager computation of unnecessary display content.

Another example of laziness in modern computer systems is copy-on-write page allocation.

See also:
non-strict programming language
functional programming
lambda calculus
combinatory logic
Currying
Graph reduction

Compare:
Lazy initialization

Parser

A parser is a computer program or a component of a program that analyses the grammatical structure of an input, with respect to a given formal grammar, a process known as parsing. Parsers can be made both for natural languages and for programming languages. Programming language parsers tend to be based on context free grammars as fast and efficient parsers can be written for them. For example LALR parsers are capable of efficiently analysing a wide class of context free grammars. Such parsers are usually not written by hand but generated by parser generators.

The task of the parser can be summarized as to determine if and how the input can be derived from the start symbol with the rules of the formal grammar. A parser can do this in essentially two ways: it can start with the input and attempt to rewrite it to the start symbol, a so-called bottom-up parser, or it can start with the start symbol and try to rewrite it to the input, a so-called top-down parser. For example LL parsers are top-down parsers and LR parsers are bottom-up parsers.

Another important distinction is whether the parser generates a leftmost derivation or a rightmost derivation (see context-free grammar). LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).

Top-down parsers
As the name suggests, a Top-down parser works in principle by constructing an abstract syntax tree from the Top node on Down, usually in a pre-order tree traversal pattern. The syntax tree is derived according to the rules of the grammar and the current input token. See the links below for common types of Top-down parsers.
Recursive descent parser
LL parser

Bottom-up parsers
LR parser
SLR parser
LALR parser
Canonical LR parser

See also
Chart parser
Compiler-compiler
CYK algorithm
Earley parser
JavaCC
Yacc
Lex

External links

Parser Robot AI Mind Module of AI4U Textbook

Lexical analysis

Lexical analysis is the process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called "lexical tokens", or just "tokens", which may be handled more easily by a parser.

A lexical analyzer, or lexer for short, typically has two stages. The first stage is called the scanner and is usually based on a finite state machine. It reads through the input one character at a time, changing states based on what characters it encounters. If it lands on an accepting state, it takes note of the type and position of the acceptance, and continues. Eventually it lands on a "dead state," which is a non-accepting state which goes only to itself on all characters. When the lexical analyzer lands on the dead state, it is done; it goes back to the last accepting state, and thus has the type and length of the longest valid lexeme.

A lexeme, however, is only a string of characters known to be of a certain type. In order to construct a token, the lexical analyzer needs a second stage. This stage, the evaluator, goes over the characters of the lexeme to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. (Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing. The evaluators for integers, identifiers, and strings can be considerably more complex. Sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments.)

For example, in the source code of a computer program the string

```
net_worth_future = (assets - liabilities);
```

might be converted (with whitespace suppressed) into the lexical token stream:

```
NAME "net_worth_future"
EQUALS
OPEN_PARENTHESIS
NAME "assets"
MINUS
NAME "liabilities"
CLOSE_PARENTHESIS
SEMICOLON
```

Lexical analysis makes writing a parser much easier. Instead of having to build up names such as "net_worth_future" from their individual characters, the parser can start with tokens and concern itself only with syntactical matters. This leads to efficiency of programming, if not efficiency of execution. However, since the lexical analyzer is the subsystem that must examine every single character of the input, it can be a compute-intensive step whose performance is critical, such as when used in a compiler.

Though it is possible and sometimes necessary to write a lexer by hand, lexers are often generated by automated tools. These tools accept regular expressions which describe the tokens allowed in the input stream. Each regular expression is associated with a phrase in a programming language which will evaluate the lexemes that match the regular expression.

The tool then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phrases, and a routine which uses them appropriately.

Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, a NAME token might be any alphabetical character or an underscore, followed by any number of instances of any alphanumeric character or an underscore. This could be represented compactly by the string [a-zA-Z_][a-zA-Z_0-9]*. This means "any character a-z, A-Z or _, then 0 or more of a-z, A-Z, _ or 0-9".

Regular expressions and the finite state machines they generate are not capable of handling recursive patterns, such as "n opening parentheses, followed by a statement, followed by n closing parentheses." They are not capable of keeping count, and verifying that n is the same on both sides -- unless you have a finite set of permissible values for n. It takes a full-fledged parser to recognize such patterns in their full generality. A parser can push parentheses on a stack and then try to pop them off and see if the stack is empty at the end.

The Lex programming language and its compiler is designed to generate code for fast lexical analysers based on a formal description of the lexical syntax. It is not generally considered sufficient for applications with a complicated set of lexical rules and severe performance requirements; for instance, the GNU Compiler Collection uses hand-written lexers.

Regular expression

A regular expression (abbreviated as regexp or regex) is a string that describes a whole set of strings, according to certain syntax rules. These expressions are used by many text editors and utilities (especially in the Unix operating system) to search bodies of text for certain patterns and, for example, replace the found strings with a certain other string.

Brief history

The origin of regular expressions lies in automata theory and formal language theory (both part of theoretical computer science). These fields study models of computation (automata) and ways to describe and classify formal languages. A formal language is nothing but a set of strings. In the 1940s, Warren McCulloch and Walter Pitts described the nervous system by modelling neurons as small simple automata. The mathematician, Stephen Kleene, later described these models using his mathematical notation called regular sets. Ken Thompson built this notation into the editor qed, then into the Unix editor ed and eventually into grep. Ever since that time, regular expressions have been widely used in Unix and Unix-like utilities such as: expr, awk, Emacs, vim, lex, and Perl. Most tools use an implementation of the regex library built by Henry Spencer.

Regular expressions in Formal Language Theory

Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. Given a finite alphabet Σ the following constants are defined:
(empty set) ∅ denoting the set ∅
(empty string) ε denoting the set {ε}
(literal character) a in Σ denoting the set {"a"}
and the following operations:
(concatenation) RS denoting the set { αβ | α in R and β in S }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
(set union) R U S denoting the set union of R and S.
(Kleene star) R* denoting the smallest superset of R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R. For example, {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "abababab", ... }.

To avoid brackets it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then brackets may be omitted. For example, (ab)c is written as abc and a U (b(c*)) can be written as a U bc*.

Sometimes the complement operator ~ is added; ~R denotes the set of all strings over Σ that are not in R. In that case the resulting operators form a Kleene algebra. The complement operator is redundant: it can always be expressed by only using the other operators.

Examples:
a U b* denotes {"a", ε, "b", "bb", "bbb", ...}
(a U b)* denotes the set of all strings consisting of 'a's and 'b's, including the empty string
b*(ab*)* the same

ab*(c U ε) denotes the set of strings starting with 'a', then zero or more 'b's and finally optionally a 'c'.
(bb U a(bb)*aa U a(bb)*(ab U ba)(bb)*(ab U ba))* denotes the set of all strings which contain an even number of 'b's and a number of 'a's divisible by three.

Regular expressions in this sense can express exactly the class of languages accepted by finite state automata: the regular languages. There is, however, a significant difference in compactness: some classes of regular languages can only be described by automata that grow exponentially in size, while the required regular expressions only grow linearly. Regular expressions correspond to the type 3 grammars of the Chomsky hierarchy and may be used to describe a regular language.

We can also study expressive power within the formalism. As the example shows, different regular expressions can express the same language: the formalism is redundant.

It is possible to write an algorithm which for two given regular expressions decides whether the described languages are equal - essentially, it reduces each expression to a minimal deterministic finite state automaton and determines whether they are isomorphic (equivalent).

To what extent can this redundancy be eliminated? Can we find an interesting subset of regular expressions that is still fully expressive? Kleene star and set union are obviously required, but perhaps we can restrict their use. This turns out to be a surprisingly difficult problem. As simple as the regular expressions are, it turns out there is no method to systematically rewrite them to some normal form. They are not finitely axiomatizable. So we have to resort to other methods. This leads to the star height problem.

Regular expression syntaxes

Traditional Unix regexps

The "basic" Unix regexp syntax is now defined as obsolete by POSIX, but is still widely used for the purposes of backwards compatibility. Most Unix utilities (grep, sed...) use it by default.

In this syntax, most characters are treated as literals - they match only themselves ("a" matches "a", "abc" matches "abc", etc). The exceptions are called metacharacters:

. Matches any single character
[ ] Matches a single character that is contained within the brackets - [abc] matches "a", "b", or "c". [a-z] matches any lowercase letter.
[^] Matches a single character that is not contained within the brackets - [^a-z] matches any single character that isn't a lowercase letter
^ Matches the start of the line
$ Matches the end of the line
\(\) Mark a part of the expression. What the enclosed expression matched to can be recalled by \n where n is a digit from 1 to 9.
\n Where n is a digit from 1 to 9; matches to the exact string what the expression enclosed in the n 'th left parenthesis and its pairing right parenthesis has been matched to. This construct is theoretically irregular and has not adopted in the extended regular expression syntax.
*

A single character expression followed by "*" matches to zero or more iteration of the expression. For example, "[xyz]*" matches to "", "x", "y", "zx", "zyx", and so on.

A \n*, where n is a digit from 1 to 9, matches to zero or more iteration of the exact string what the expression enclosed in the n 'th left parenthesis and its pairing right parenthesis has been matched to. For example, "\(a??\)\1" matches to "abcbc" and "adede" but not "abcde".

An expression enclosed in "\(" and "\)" followed by "*" is deemed to be invalid. In some cases (e.g. /usr/bin/xpg4/grep of SunOS 5.8), it matches to zero or more iteration of the same string which the enclose expression matches to. In other some cases (e.g. /usr/bin/grep of SunOS 5.8), it matches to what the enclose expression matches to, followed by a literal "*".
\{x,y\} Match the last "block" at least x and not more than y times. - "a\{3,5\}" matches "aaa", "aaaa" or "aaaaa".

There is no representation of the set union operator in this syntax.

Examples:
".at" matches any three-letter word ending with "at"
"[hc]at" matches "hat" and "cat"
"[^b]at" matches any three-letter word ending with "at" and not beginning with 'b'.
"^[hc]at" matches "hat" and "cat" but only at the beginning of a line
"[hc]at$" matches "hat" and "cat" but only at the end of a line

POSIX modern (extended) regexps

The more modern "extended" regexp can often be used with modern Unix utilities by including the command line flag "-E".

POSIX extended regexps are similar in syntax to the traditional Unix regexp, with some exceptions. The following metacharacters are added:

+ Match the last "block" one or more times - "ba+" matches "ba", "baa", "baaa" and so on
? Match the last "block" zero or one times - "ba?" matches "b" or "ba"
| The choice (or set union) operator: match either the expression before or the expression after the operator - "abc|def" matches "abc" or "def".

Also, backslashes are removed: \{...\} becomes {...} and \(...\) becomes (...)

Examples:
"[hc]+at" matches with "hat", "cat", "hhat", "chat", "hcat", "ccat" et cetera
"[hc]?at" matches "hat", "cat" and "at"
"([cC]at | [dD]og)" matches "cat", "Cat", "dog" and "Dog"

Since the characters '(', ')', '[', ']', '.', '*', '?', '+', '^' and '$' are used as special symbols they have to be "escaped" somehow if they are meant literally. This is done by preceding them with '\' which therefore also has to be "escaped" this way if meant literally.

Examples:
".\.(\(|\))" matches with the string "a.)"

Perl Compatible Regular Expressions (PCRE)

Perl has a much richer syntax than even the extended POSIX regexp. This syntax has also been used in other utilities and applications -- exim, for example. Although still named "regular expressions", the Perl extensions give an expressive power that far exceeds the regular languages.

This extra power comes at a cost. The worst-case complexity of matching a string against a Perl regular expression is exponential in the size of the input. That means the regular expression could take an extremely long time to process under just the right conditions of expression and input string, although it rarely happens in practice.

External links

The Regulator: Advanced and free Regex testing tool
Regexlib.com: online regular expression archive
Regular expression information
Syntax and Semantics of Regular Expressions by Xerox Research Centre Europe
Regexp Syntax Summary, reference table for different styles of regular expressions
Learning to Use Regular Expressions by David Mertz
The Regex Coach, a very powerful interactive learning system to hone your regexp skills
Mastering Regular Expressions book website
Regenechsen Beginners tutorial for using Regular Expressions

Context-free grammar

In computer science a context-free grammar (CFG) is a formal grammar in which every production rule is of the form
$V \rightarrow w$
where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. The term "context-free" comes from the feature that the variable V can always be replaced by w, in no matter what context it occurs. A formal language is context-free if there is a context-free grammar which generates it.

Context-free grammars are important because they are powerful enough to describe the syntax of programming languages; in fact, almost all programming languages are defined via context-free grammars. On the other hand, context-free grammars are simple enough to allow the construction of efficient parsing algorithms which for a given string determine whether and how it can be generated from the grammar. See Earley parser, LR parser and LL parser.

BNF (Backus-Naur Form) is often used to express context-free grammars.

Examples

Example 1

A simple context-free grammar is
$S \rightarrow aSb \mid \varepsilon$
where | is used to separate different options for the same non-terminal and ε stands for the empty string. This grammar generates the language which is not regular.

Example 2

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:
$S \rightarrow x \mid y \mid z \mid S + S \mid S * S \mid S - S \mid S/S \mid (S)$

This grammar can for example generate the string "(x + y) * x - z * y / (x + x)".

Example 3

A context-free grammar for the language consisting of all strings over {a,b} which contain a different number of a's than b's is
$S \rightarrow U \mid V$
$U \rightarrow TaU \mid TaT$
$V \rightarrow TbV \mid TbT$
$T \rightarrow aTbT \mid bTaT \mid \varepsilon$
Here, T can generate all strings with the same number of a's as b's, U generates all strings with more a's than b's and V generates all strings with fewer a's than b's.

Other examples

Context-free grammars are not limited in application to mathematical ("formal") languages. The ancient Indian linguist Panini described Sanskrit using a context-free grammar. Recently, it has been suggested that a class of Tamil poetry called Venpa is governed by a context-free grammar.

Derivations and Syntax trees

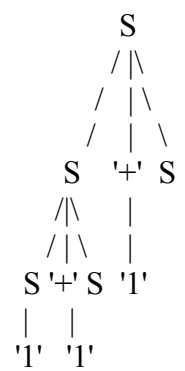
There are basically two ways to describe how in a certain grammar a string can be derived from the start symbol. The simplest way is to list the consecutive strings of symbols, beginning with the start symbol and ending with the string, and the rules that have been applied. If we introduce a strategy such as "always replace the left-most nonterminal first" then for context-free grammars the list of applied grammar rules is by itself sufficient. This is called the leftmost derivation of a string. For example, if we take the following grammar:
(1) S → S + S
(2) S → 1
and the string "1 + 1 + 1" then the left derivation of this string is the list [(1), (1), (2), (2), (2)]. Analogously the rightmost derivation is defined as the list that we get if we always replace the rightmost nonterminal first. In this case this would be the list [(1), (2), (1), (2), (2)].

The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed. See for an example LL parsers and LR parsers.

A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example the structure of the string "1 + 1 + 1" would, according to the leftmost derivation, be:

S→S+S (1)
S→S+S+S (1)
S→1+S+S (2)
S→1+1+S (2)
S→1+1+1 (2)

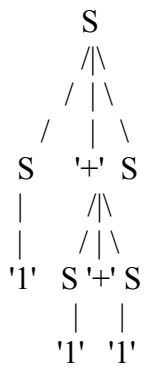
{ { { 1 }S + { 1 }S }S + { 1 }S }S
where { ... }S indicates a substring recognized as belonging to S. This hierarchy can also be seen as a tree:



This tree is called a concrete syntax tree (see also abstract syntax tree) of the string. In this case the presented leftmost and the rightmost derivation define the same syntax tree, however there is another (leftmost) derivation of the same string possible

S→ S + S (1)
S→ 1 + S (2)
S→ 1 + S + S (1)
S→ 1 + 1 + S (2)
S→ 1 + 1 + 1 (2)

and this defines the following syntax tree:



If for certain strings in the language of the grammar there are more than one parsing tree then the grammar is said to be an ambiguous grammar. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply.

Normal forms

Every context-free grammar which does not generate the empty string can be transformed into an equivalent one in Chomsky normal form or Greibach normal form. "Equivalent" here means that the two grammars generate the same language.

Because of the especially simple form of production rules in Chomsky Normal Form grammars, this normal form has both theoretical and practical implications. For instance, given a context-free grammar, one can use the Chomsky Normal Form to construct a polynomial-time algorithm which decides whether a given string is in the language represented by that grammar or not (the CYK algorithm).

Properties of context-free languages

An alternative and equivalent definition of context-free languages employs non-deterministic push-down automata: a language is context-free if and only if it can be accepted by such an automaton. The union and concatenation of two context-free languages is context-free; the intersection need not be. The reverse of a context-free language is context-free, but the complement need not be. Every regular language is context-free because it can be described by a regular grammar. The intersection of a context-free language and a regular language is always context-free. There exist context-sensitive languages which are not context-free.
To prove that a given language is not context-free, one employs the pumping lemma for context-free languages.

Byte-code

Byte-code is a sort of intermediate code that is more abstract than machine code. It is often treated as a binary file containing an executable program much like an object module, which is a binary file containing a machine code produced by compilers.

Byte-code is called so because usually each op code is one-byte length but the length of instruction code varies. Each instruction has one byte operation code from 0 to 255 followed by parameters such as registers or memory address. This is a typical case, but the specification of bytecode largely varies in language.

As in intermediate code, it is a form of output code used by programming language implementors to reduce dependence on specific hardware and ease interpretation.

Less commonly, bytecode is used as an intermediate code of a compiler. Some systems, called dynamic translators, or "just-in-time" (JIT) compilers, translate bytecode into machine language immediately prior to runtime to improve execution speed.

A byte-code program is normally interpreted by a byte-code interpreter (usually called virtual machine since it is like a computer machine). The advantage is portability, that is, the same binary code can be executed across different platforms or architectures. This is the same advantage as that of interpreted languages. However, because bytecode is usually less abstract, more compact, and more computer-centric than program code that is intended for human modification, the performance is usually better than mere interpretation. Because of its performance advantage, today many interpreted languages are actually compiled into bytecode once then executed by bytecode interpreter. Such languages include Perl and Python. Java code is typically transmitted as bytecode to a receiving machine, which then uses a JIT compiler to translate the bytecode to machine code before execution. The current implementation of the Ruby programming language actually does not use bytecode, instead, it relies on tree-like structures, which resembles intermediate representation used in compilers.

Also of interest are p-Codes, which are just like byte codes, but may be physically larger than a single byte and may vary in size (much like Opcodes do). They work at very high levels, such as "print this string" and "clear the screen". Both BASIC and some versions of Pascal use p-Codes.

Examples
O-code of the BCPL programming language
p-Code of UCSD Pascal implementation of the Pascal programming language
Bytecodes of many implementations of the Smalltalk programming language
Java byte code, which is executed by the Java virtual machine.

Complexity Metrics and Models

The Software Science [Halstead 77] developed by M.H.Halstead principally attempts to estimate the programming effort. The measurable and countable properties are (taken from http://yunus.hun.edu.tr):

n1 = number of unique or distinct operators appearing in that implementation
n2 = number of unique or distinct operands appearing in that implementation
N1 = total usage of all of the operators appearing in that implementation
N2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

1. The program vocabulary as n = n1 + n2
2. The program implementation length as N = N1 + N2

Operators can be "+" and "*" but also an index "[...]" or a statement separation "..;..". The number of operands consists of the numbers of literal expressions, constants and variables.

Length Equation

It may be necessary to estimate the relationship between length N and vocabulary n. A prime on N (and other symbols later on) means it is calculated rather than counted:

N' = n1 log2(n1) + n2 log2(n2)

It is experimentally observed that N ' gives a rather close agreement to program length.

Quantification of Intelligence Content

The same algorithm needs more consideration in a low level programming language. It is easier to program in Pascal rather than in assembly. The intellegence Content determines how much is said in a program. In order to find Quantification of Intelligence Content we need some other metrics and formula:

Program Volume is the minimum number of bits required for coding the program. This metric is for the size of any implementation of any algorithm.

V = N log2(n)

Program Level at which the program can be understood. It is the relationship between Program Volume (V) and Potential Volume (V'). Only the simplest algorithm can have a level of unity.

L = V' / V

Program Level Equation is used when the value of Potential Volume is not known.

L' = 2 n2 / (n1 N2)

Level of difficulty in the program:
D’ = 1 / L’

Intelligence Content

I = L’ V = V / D’

In this equation all terms on the right-hand side are directly measurable from any expression of an algorithm. The intelligence content is correlated highly with the potential volume. Consequently, because potential volume is independent of the language, the intelligence content should also be independent.

Programming Effort

The programming effort is restricted to the mental activity required to convert an existing algorithm to an actual implementation in a programming language. In order to find Programming effort we need some metrics and formulas:

Potential Volume : is a metric for denoting the corresponding parameters in an algorithm's shortest possible form. Neither operators nor operands can require repetition.

V’ = (n*1 + n*2) log2(n*1 + n*2)

Effort Equation: the total number of elementary mental discriminations is :

E = V / L’ = V**2 / V’ = D’ V

If we express it: The implementation of any algorithm consists of N selections (nonrandom > of a vocabulary n. a program is generated by making as many mental comparisons as the program volume equation determines, because the program volume V is a measure of it. Another aspect that influences the effort equation is the program difficulty. Each mental comparison consists of a number of elementary mental discriminations. This number is a measure for the program difficulty.

Time Equation

A concept concerning the processing rate of the human brain, developed by the psychologist John Stroud, can be used. Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination. The Stroud number S is then Stroud's moments per second with 5 <= S <= 20. Thus we can derive the time equation where, except for the Stroud number S, all of the parameters on the right are directly measurable:

T’ = D’ log2(n) / (N’ S)

Advantages of Halstead:

- Do not require in-depth analysis of programming structure.
- Predicts rate of error.
- Predicts maintenance effort.
- Useful in scheduling and reporting projects.
- Measure overall quality of programs.
- Simple to calculate.

Can be used for any programming language.

Numerous industry studies support the use of Halstead in predicting programming effort and mean number of programming bugs.

Drawbacks of Halstead:

It depends on completed code.

It has little or no use as a predictive estimating model. But McCabe's model is more suited to application at the design level.

McCabe's Cyclomatic number

A measure of the complexity of a program was developed by [McCabe 1976]. He developed a system which he called the cyclomatic complexity of a program. This system measures the number of independent paths in a program, thereby placing a numerical value on the complexity. In practice it is a count of the number of test conditions in a program. The cyclomatic complexity (CC) of a graph (G) may be computed according to the following formula:

CC(G) = Number (edges) - Number (nodes) + 1

The results of multiple experiments (G.A. Miller) suggest that modules approach zero defects when McCabe's Cyclomatic Complexity is within 7 ± 2. A study of PASCAL and FORTRAN programs (Lind and Vairavan 1989) found that a Cyclomatic Complexity between 10 and 15 minimized the number of module changes. Thomas McCabe who is the invitor of cyclomatic complexity has founded a metrics company, McCabe & Associates There are some other metrics that are inspritted from cyclomatic complexity. You can find them at McCabe metrics

Advantages of McCabe Cyclomatic Complexity:

It can be used as a ease of maintenance metric.

Used as a quality metric, gives relative complexity of various designs.

It can be computed early in life cycle than of Halstead's metrics.

Measures the minimum effort and best areas of concentration for testing.

It guides the testing process by limiting the program logic during development.

Is easy to apply.

Drawbacks of McCabe Cyclomatic Complexity :

The cyclomatic complexity is a measure of the program's control complexity and not the data complexity

The same weight is placed on nested and non-nested loops. However, deeply nested conditional structures are harder to understand than non-nested structures.

It may give a misleading figure with regard to a lot of simple comparisons and decision structures. Whereas the fan-in fan-out method would probably be more applicable as it can track the data flow

Fan-In Fan-Out Complexity - Henry's and Kafura's

Henry and Kafura (1981) [from Sommerville 1992] identified a form of the fan in - fan out complexity which maintains a count of the number of data flows from a component plus the number of global data structures that the program updates. The data flow count includes updated procedure parameters and procedures called from within a module.

Complexity = Length x (Fan-in x Fan-out)**2

Length is any measure of length such as lines of code or alternatively McCabe's cyclomatic complexity is sometimes substituted. Henry and Kafura validated their me tric using the UNIX system and suggested that the measured complexity of a component allowed potenetially faulty system components to be identified. They found that high values of this metric were often measured in components where there had historically been a high number of problems.

Advantages of Henry's and Kafura's Metic

It takes into account data-driven programs

It can be derived prior to coding, during the design stage

Drawbacks of Henry's and Kafura's Metic

It can give complexity values of zero if a procedure has no external interactions