Programming Languages*

October 1, 2014

Disclaimer

These notes have been prepared with the **only** purpose to help me pass the Computer Science qualifiying exam at the University of Illinois at Chicago. They are distributed as they are (including errors, typos, omissions, etc.) to help other students pass this exam (and possibly relieving them from part of the pain associated with such a process). I take **no responsibility** for the material contained in these notes (which means that you can't sue me if you don't pass the qual!) Moreover, this pdf version is distributed together with the original LaTeX (and LyX) sources hoping that someone else will improve and correct them. I mean in absolute no way to violate copyrights and/or take credit stealing the work of others. The ideas contained in these pages are **not mine** but I've just aggregated information scattered all over the internet.

Contents

1 Important programming paradigms

1.1 Procedural PLs (Imperative)

Sample languages: C, ADA, Pascal, ALGOL, ...

Based on procedures (functions). Modularity.

1.2 Object Oriented PLs (Imperative)

Sample languages: COBOL, Simula, ADA, Smalltalk, C++, Java, Eiffel...

They are based on the use of units called objects which are a mixture of data and methods to manipulate them. Core features of OOP are: *encapsulation*, *inheritance*, *polymorphism*, *dynamic dispatch* (runtime decision on the method to execute) and *open recursion* (this keyword).

1.3 Functional (Declarative)

Sample languages: LISP, ML, Haskell, Erlang,...

They try to mimic mathematical functions and therefore they DON'T make use of variables "freeing" the programmer from details related to the memory space at execution (however this makes them extremely inefficient). This implies there are NO loops and repetition is achieved through recursion. Programming is reduced to writing functions while execution consists of an evaluation of those functions.

Referential transparency: evaluation of the same function with the same parameters always produces the same result.

They provide a set of primitive functions, a set of functional forms (i.e. composition) to construct more complex functions (from the primitive ones), a function application operation and some structures to hold the data.

1.4 Logic (Declarative)

Sample languages: Prolog,...

Their approach is to express programs in form of symbolic logic and use a logical inferencing process to produce results. Since they are declarative only the specifications of the desired results are stated rather than detailed procedures to produce them.

2 Binding

Programs deal with *entities*: variables, subprograms, expressions, statements, etc. All program entities have propertied, called *attributes*, which characterize them.

In a general sense, a *binding* is an association between an *attribute* (name, scope, value, type,...) and an *entity* (function, variable,...) or between an operation and a symbol.

The moment when this association takes place is called *binding time*. Bindings can take place at language design time (e.g. meaning of * symbol), language implementation time (e.g. name of a variable), compile time (e.g. overloading of + symbol), link time, load time or run time (e.g. value of a variable).

Also, binding can, in some cases, never happen! (e.g. variable type in assembly) In such a case, it is useful to see the unbound parameter as optional.

2.0.1 Static & dynamic binding

- A binding is *static* if it occurs before run time and remains unchanged throughout program execution.
- If a binding first occurs during run time or can change in the course of program execution, it is called *dynamic*.

3 Variables

A program *variable* is an abstraction of a computer memory cell or collection of cells. A variable can be characterized as a sextuple:

Name: the string used to identify the variable.

Type: determines the range of values the variable can have and the set of operations that are defined for values of the type.

Scope: portion of program where the variable is visible.

Lifetime: time during which the variable is bound to a memory location.

Address (or l-value): it's the memory location associated with the variable. When more than one variable name can be used to access a single memory location, the names are called *aliases*. (see overloading)

Value (or r-value): it's the content of the memory cell (or cells) associated with the variable.

A detailed explanation of each one of these elements follows.

3.1 Names

A *name* is a string of characters used to identify some entity in a program.

3.1.1 Overloading

We say a name is overloaded when the same name refers to two different entities. However, the specific occurrence provides enough information for the binding to be unambiguous.

Not to be confused with aliasing where two names refer to the same entity. (see l-value and r-value below)

3.2 Data types

The *data type* of a variable determines the range of values the variable can have and the set of operations that are defined for values of the type. Before a variable can be referenced in a program, it must be bound to a data type. The two important aspects of this binding are *how* the type is specified and *when* the binding takes place.

Note that a language can be also *typeless*: type is an abstraction which is useful but not necessary.

Each language provides some p*rimitive data types* (which usually include: numeric types, Boolean types and character types) and constructs to assemble user-defined data types.

3.2.1 Static typing through variable declaration

An *explicit declaration* is a statement in a program that lists variable names and specifies that they are a particular type. An *implicit declaration* is a means of associating variables with types through default conventions instead of declaration statements. In this case, the first appearance of a variable name in a program constitutes its implicit declaration.[1]

Both explicit and implicit declarations create *static bindings* to types.

3.2.2 Dynamic typing

The variable is bound to a type when it is assigned a value in an assignment statement. When the assignment statement is executed, the variable being assigned is bound to the type of the value, variable, or expression on the right side of the assignment.

3.2.3 Comparison between the two strategies

The primary advantage of dynamic over static binding of variables to types is that it provides a great deal of *programming flexibility.*

There are two disadvantages to dynamic type binding.

First, the error detection (type checking) capability of the compiler is diminished compared to a compiler for a language with static bindings, because any two types can appear at opposite sides of the assignment operator.

Second, the cost of implementing dynamic attribute binding is considerable, particularly in execution time, since type checking must be done at run time.

[1] In C and C++, we must sometimes distinguish between declarations and definitions. Declarations specify types and other attributes but do not cause allocation of storage. Definitions specify attributes and cause storage allocations.

3.3 Type checking

Type checking is the activity of ensuring that the operands of an operator (including subprograms and assignment statements) are compatible types. A *compatible* type is one that is either legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type. This automatic conversion is called a *coercion*. A type error is the application of an operator to an operand of an inappropriate type.

3.3.1 Strong typing

We define a programming language to be *strongly typed* if type errors are always detected. This requires that:

- each name in a program in the language has a single type associated with it, and that type is known at compile time. (i.e. all the types are statically bound)
- the types of all operands can be determined, either at compile time or at run time.

3.3.2 Compatibility

There are two different compatibility methods:

Name type compatibility means that two variables have compatible types only if they are in either the same declaration or in declarations that use the same type names. This is easy to implement but is highly restrictive.

Structure type compatibility means that two variables have compatible types if their types have identical structures. This is more flexible but is more difficult to implement.

3.4 Scope

The *scope* of a variable is the range of statements over which it is visible. A variable is visible in a statement if it can be referenced in that statement. A *referencing environment* is the collection of all names that are visible in the statement.

We have two types of scope:

Static: their scope of the variable can be determined before the execution starts. The referencing environment for a statement is made of all the local variables plus all the variables in the static ancestors.

Cons:	too many functions/data are visible to children procedures, hard restructuring of code
Pros:	easy to read, local variable hidden, higher reliability

Dynamic: the scope is determined by the calling sequence of subprograms. The referencing environment for a statement is made of all the local variables plus all the visible variables in all active subprograms. (a subprogram is active if its execution has begun but has not yet terminated)

Cons:	no static type check, low readability, longer access to non-locals (chain lookup)
Pros:	variables in the caller are implicitly visible in the called

Note that, for both static and dynamic scoping, the most specific variable overrides the less specific.

3.5 Lifetime

The *lifetime* of a variable is the time during which it is bound to a particular memory cell, that is, the time between it's *allocation* and *deallocation*.

To investigate storage bindings of variables, it is convenient to separate scalar variables into four categories:

Static Static variables are those that are bound to a memory cell before program execution begins. (`static` variables in C, C++ and Java).

Stack-dynamic Stack-dynamic variables are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. (local variables in C)

Explicit Heap-dynamic Explicit Heap-dynamic variables are nameless (abstract) memory cells that are allocated and deallocated by explicit directives, specified by the programmer. These variables, which are allocated from and deallocated to the heap, can only be referenced through pointer or reference variables (objects in Java, `new` and `delete` in C++)

Implicit Heap-dynamic: Implicit Heap-dynamic variables are bound to heap storage only when they are assigned values. In fact, all their attributes are bound every time they are assigned: in a sense, they are just names that adapt to whatever use they are asked to serve. (variables in JavaScript and all the major scripting languages)

3.5.1 Scoping and lifetime

The scope and the lifetime of a variable are clearly not the same because static scope is a textual, or spatial, concept whereas lifetime is a temporal concept. This means you can't compare them!

3.6 l-value and r-value

The *l-value* of a variable (or its *address*) is the memory address (i.e. the storage area) with which it is associated.

The *r-value* (or simply *value*) is the actual data that is stored in such a memory location.

3.6.1 Aliases

When more than one variable name can be used to access a single memory address, the names are called *aliases*. Aliases decrease readability of programs and allow variables to be changed by manipulation of other variables.

4 Subprograms

A *subprogram* (also called *subroutine*, *procedure*, *method*, *function*, or *routine*) is a portion of code within a larger program that performs a specific task and is relatively independent of the remaining code. A subprogram can be characterized as a 5-tuple:

Name: the string used to identify the subprogram.

Scope: generally each subprogram has its scope (for other statements to reference them) and is also allowed to define its own variables (local variables), thereby defining its own local referencing environment.

l-value: it's the memory location associated with the subprogram. This is used by those languages that allow pointers to subprograms.

r-value: the executable statements of the subprogram.

Signature: the input parameters and return types of the subprogram.

4.1 Parameter passing

The parameters in the subprogram header are called *formal parameters*. When a subprogram is called, a list of parameters to be bound to the formal parameters of the subroutine, is required: these are called *actual parameters*. There are two ways the binding can take place:

positional parameters, in which the first actual parameter is bound to the first formal parameter and so on, and

keyword parameters, where the name of the formal parameter to which an actual parameter is to be bound, is specified with the actual parameter.

There are three semantics models to transfer parameters to a subprogram: *in mode*, *out mode* and *in-out mode*. There are also two conceptual models of how data transfers take place: either the *actual value* is physically moved or an *access path* to the data is transmitted. The previous can be combined in various ways, yielding to the following parameter passing methods:

Pass-by-value: the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram. Main disadvantages: could require expensive physical move. Also, if a function is passed as parameter it needs to be evaluated and, if the function never returns, the call will never be executed.

Pass-by-result: the formal parameter acts as a local variable and, right before the control is transferred back to the caller, its value is passed back to the caller. Main disadvantages: same as pass-by-value, assignment ordering problem (e.g. `sub(p1,p1)`)

Pass-by-value-result: combination of the previous two...

Pass-by-reference: transmits an access path, usually the variable address, to the called program.[2] Main disadvantages: aliasing!!! Slower accesses to value due to additional layer in addressing.[3]

Pass-by-name: the actual parameters are textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. Semantic varies based on the type of the parameter (scalar variable, constant, array element, arithmetic expression, etc.). Main disadvantages: slowness of textual substitution, confusing.

Subprogram calls can be type checked, meaning that the type of each actual parameter is checked against the type of the corresponding formal parameter.

4.1.1 Passing parameters that are subprogram names

A first problem is how to type check the parameters of the activations of the subprogram that was passed as parameter.

A second problem is how to determine the correct referencing environment for executing the passed subprogram. There are three possible strategies:

Shallow binding: use the environment of the call statement that enacts the passed subprogram (i.e. the environment at the point where the subprogram is executed inside the subprogram that takes it as a parameter).

Deep binding: use the environment of the definition of the passed subprogram (i.e. the environment at the point where the definition of the actual passed subprogram is)

Ad hoc binding: use the environment of the call statement that passed the subprogram as an actual parameter

[2] We can think of this as a "shared variable" between the caller and the called.

[3] Note how this method produces different results from pass-by-value-result when: (a) two formal parameters become aliases; (b) a formal parameter and a non local variable become aliases.

4.1.2 Generic subprograms

A *generic* or *polymorphic subprogram* takes parameters of different types on different activations. *Parametric polymorphism* is provided by subprograms that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram. (e.g. templates in C++)

See Section 7.4 for more details on polymorphism.

4.2 Declaration vs Definition

Subprograms can have *declarations*, also called *prototypes*, (used to provide type information but not to declare variables) as well as *definitions* (where the body of the subprogram is specified). They are necessary when the compiler must translate a call to a subprogram before it has seen that subprogram's definition. A classic example of this is *recursion*.

5 Runtime environment

A subprogram consists of two separate parts: the *code segment* (i.e. the actual code of the subprogram), which is constant, and the *activation record* (or *frame*), which consists of all the data that can change when the subprogram is executed. An *activation record instance* (ARI) is a concrete example of an activation record.

5.1 Static languages (FORTRAN)

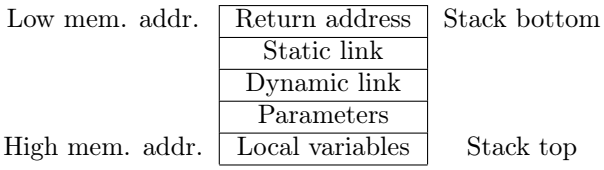
In static languages, memory requirements can be evaluated before program execution and, since activation records have fixed size, they can be statically allocated (i.e. before the execution starts). Recursion, in such languages, is clearly impossible.

5.2 Stack languages (ALGOL-like)

In ALGOL-like languages, memory requirements can be predicted and the execution is organized with a LIFO discipline.

The code resides in a separate part of the memory and it is executed one instruction after the other according to the instructions themselves. The address of the currently executing instruction is stored in the *instruction pointer* register (*ip*). This way we can execute the same code multiple times with different activation records, supporting recursion.

The format of an activation record for a given subprogram in a static scoped language is known at compile time and, typically, it is in the form shown below.



The return address, static link, dynamic link and parameters are placed on the activation record by the calling function so they must come first.

Return address: it is a pointer to the code segment of the caller and an offset address offset in that code segment containing the instruction following the call.

Static link: points to the bottom of an ARI of the static parent and is used to reference non local variables. (see 5.2.2)

Dynamic link: point to the top of the ARI of the caller. This pointer is used in the destruction of the current ARI when the procedure completes its execution. The stack top (*stack pointer* or *sp*) is set to the value of the old dynamic link.

Parameters: the actual parameters in the ARI are the values or the addresses provided by the caller.

Local variables: local variables of the called subprogram. References to such variables can be represented in the code as offsets from the beginning of an activation record. Such an offset is called *local offset*.

Activating a subprogram requires the dynamic creation of its ARI (i.e. the binding to a *base address* of the ARI). The reason this must be created at run-time is that, even if the form of an activation record is known, the size of it is unknown at compile time.

5.2.1 Recursion

When recursion comes into play we need to add a new entry to the activation record (in the top position) called *functional value*, used to store the returned value of the recursive function.

5.2.2 Nonlocal references

A reference to a non local variable requires a two-step access process. The first one is to find the ARI in the stack in which the variable was allocated. The second step is to use the *local offset* of the variable (within the ARI) to access it.

A *static chain* is a chain of static links that connect certain ARIs in the stack. The purpose of this chain is to implement the access to nonlocal variables in static-scoped languages.

Because the nesting of scope is known at compile time, the compiler can determine the *static depth*, which is an integer associated with a static scope that indicates how deeply it is nested in the outermost scope. The length of the static chain needed to reach the correct activation record instance for nonlocal reference to a variable `x` is exactly the difference between the static depth of the subprogram containing the reference to `x` and the static depth of the procedure containing the declaration for `x`. The actual reference could be represented as a pair (chain offset, local offset), where chain offset is clearly the numbers of links to the correct ARI.

5.2.3 Dynamic scoping

The mechanism is the same explained in the previous paragraph but instead of following the static links we follow the dynamic ones.

5.3 Heap languages

In heap languages the memory usage is unpredictable, since all data are explicitly and dynamically allocated in the heap (which grows in the opposite direction with respect to the stack).

The lifetime of such data does not depend on the unit in which their allocation statement appears but instead it lasts as long as they are referred some existing pointer variables.

6 Other entities

6.1 Expressions

A *short-circuit evaluation* of an expression is one in which the result is determined without evaluating all of the operands and/or operators. (e.g. the value of the arithmetic expression `(13 * a) * (b / 13 - 1)` is independent of the value of `(b / 13 - 1)` if `a = 0`, because `0 * x = 0` for any `x`)

6.2 Pointers

A pointer is a variable whose r-value is the l-value of another variable. The r-value of a pointer is called *reference*[4]. Pointers can be:

[4]Note that, in C++, pointers and references are different! A reference is an alias for an object and, when bound to an object, it cannot be made to refer to a different object. When passed as a parameter, a reference behaves similarly to a pointer, but the actual address of the references object is masked.

typed: requires the variable referenced by the pointer to be of a specific type.

untyped: pointers are concerned only with addresses in memory.

The fundamental operation on pointer is *dereferencing*. Dereferencing a pointer means to access the r-value of the variable referenced by the pointer.

Some languages, like C, require typed pointer but allow "pointers arithmetic" (other operations on pointers other than dereferencing) which can potentially hinder pointers type safety.

Pointers are very useful when efficiency is crucial but they are a low-level construct, and, as such, extremely error-prone.

6.2.1 Dangling pointers

A *dangling pointer* is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

6.2.2 Lost heap-dynamic variables (i.e. memory leaks)

A *lost heap-dynamic variable* is an allocated heap-dynamic variable that is no longer accessible to the user program. Such variables are often called garbage, because they are not useful for their original purpose, and they also cannot be reallocated for some new use by the program. They therefore "consume" memory without releasing it, causing what is called *memory leak*.

6.2.3 Garbage collection

By making manual memory deallocation unnecessary (and often forbidding it), *garbage collection* (GC) frees the programmer from manually dealing with memory deallocation. Garbage collection solves both dangling pointers and lost heap-dynamic variables problems (and double free bugs, very common in C). The basic principles of garbage collection are:

1. Find data objects in a program that cannot be accessed in the future
2. Reclaim the resources used by those objects

There are two distinct techniques that can be used to implement GC:

- The *reference counter method* (or *eager approach*) maintains a counter in every cell, which stores the number of pointers that are currently pointing at the cell. Embedded in the decrement operation for the reference counters, which occurs when a pointer is disconnected from the cell, is a check for a zero value. If the reference counter reaches zero, it mean that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space.
- The *lazy approach* allows the run-time system to allocate storage cells as requested and disconnects pointers from cells as necessary, without regard for storage reclamation (allowing garbage to accumulate), until it has allocated all available cells. At this point, a garbage collection process marks every heap cell as garbage (all heap cell have an extra indicator bit or field). Then, starting from the stack, every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage. Finally, all cells in the heap that have not been specifically marked as still being used are returned to the list of the available space.

7 Object-oriented related languages

A language that is OO must provide support for three key language features: abstract data types, inheritance, and a particular kind of dynamic binding of messages to methods.

7.1 Abstract Data Types (ADT)

An ADT is a data type that satisfies the following two propertied:

- The representation, or definition, of the data type and the operations on objects of the data type are contained (encapsulated) in a single syntactic unit. Also, other program units may be allowed to create variables of the defined type.
- The representation of objects of the type is hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the type's definition.

In OO languages ADTs are usually represented by classes.

7.2 Dynamic dispatch and polymorphism

The subprograms that define the operations on objects of a class are called *methods*. The calls to methods are often called *messages*. The entire collection of methods of an object is called the *message protocol*, or *message interface*, of the object.

The difference between procedures in general and an object's method is that the latter, being associated with a particular object, may access or modify the data private to that object in a way consistent with the intended behavior of the object. Consequently, rather than thinking "a method is just a sequence of commands", a programmer using an object-oriented language will consider a method to be "an object's way of providing a service" (its "method of doing the job", hence the name); a method call is thus considered to be a request to an object to perform some task. Consequently, method calls are often modelled as a means of passing a message to an object. Rather than directly performing an operation on an object, a message (most of the time accompanied by parameters) is sent to the object telling it what it should do. The object either complies or raises an exception describing why it cannot do so.

Usually this is implemented using *dynamic dispatch* which is the process of mapping a message to a specific sequence of code (method) at runtime. Normally in a typed language (Smalltalk) the dispatch mechanism will be performed based on the type of the arguments (most commonly based on the type of the receiver of a message) or use *duck typing*. Languages with weak or no typing systems often carry a dispatch table as part of the object data for each object. This allows instance behaviour as each instance may map a given message to a separate method. Some languages offer a hybrid approach. Dynamic dispatch will always incur an overhead so some languages (C++) offer the option to turn dynamic dispatch off for particular methods.

7.3 Inheritance

Inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviors which can be based on previously created ones. This is different from polymorphism since the first one is code related while the second one is data type related.

7.3.1 Overriding methods

In addition to inheriting entities from its parent class, a derived class can add new entities and modify inherited methods. A modified method has the same name, and often the same protocol, as the one pf which it is a modification. The new method is said to *override* the inherited version, which is then called *overridden method*.

7.3.2 Multiple inheritance vs single inheritance

If a class is a subclass of a single parent class, then the derivation process is called *single inheritance*. If a class has more than one parent class, the process is called *multiple inheritance*. When a number of classes are related through single inheritance, their relationship to each other can be shown in a *derivation tree*. The class relationships in a multiple inheritance can be shown in a *derivation graph*.

One obvious problem of multiple inheritance is name collision. A variation of such a problem is called *diamond inheritance*. Moreover the use of multiple inheritance can easily lead to intricate program organizations.

7.4 Polymorphism and dynamic binding

The term polymorphic is from the Greek, meaning "having multiple forms." It is applied to code, both data structures and subroutines that can work with values of multiple types. For this concept to make sense, the types must generally have certain characteristics in common, and the code must not depend on any other characteristics. The commonality is usually captured in one of two main ways.

1. In *parametric polymorphism* the code takes a type (or set of types) as a parameter, either explicitly or implicitly.

2. In *subtype polymorphism* the code is designed to work with values of some specific type T, but the programmer can define additional types to be extensions or refinements of T, and the polymorphic code will work with these subtypes as well. (a.k.a. *Liskov substitution principle*)

Explicit parametric polymorphism is also known as *genericity*. Generic facilities appear in Ada, C++ (*templates*) and recent versions of Java and C#, among others. Implicit parametric polymorphism appears in the Lisp and ML families of languages, and in various scripting languages, and has the major disadvantage of delaying type checking until run time.

Subtype polymorphism is, instead, fundamental to object-oriented languages, in which subtypes (classes) are said to inherit the methods of their parent types. This is usually implemented by creating a single copy of the code, and by inserting sufficient "metadata" in the representation of objects that the code can tell when to treat them differently.