Some Points on Principles of Programming Languages

Dick Grune
dick@cs.vu.nl

☞ This list is in no way intended to replace the book, which in its entirety constitutes the exam material. It is just a list of subjects which may require more attention than the other subjects.

Aspects of Programming Languages

- A paradigm is a coherent set of methods found useful for solving problems in a given problem domain.
- A paradigm is characterized by a principle; this supplies the basis for a limited number of concepts, which allow infinitely many methods to be invented, which in their turn help the programmer in solving the problem at hand.
- Grammar: describes the form of programs; semantics is attached to grammatical notions.
- Syntactic structure: given the program text, the grammar defines a parsing, which recovers the notions and therewith the semantics.
- Abstraction: to ignore irrelevant detail in a safe way. This is done by hiding information behind an interface.

Imperative Languages

- Imperative programming: specify everything: data declarations, state-changing and flow of control.
- Aliasing binds more than one name to one item, overloading binds one name to more than one item. When using an overloaded name, the context determines which item it refers to: the context disambiguates.
- A type is a set of values; often a set of operators is defined on it and sometimes an allocator and a destructor is defined for it.
- There are basic types (integer, etc.); more complex types can be made using type constructors.
- There is one type constructor that uses no other types: enumeration.
- There are four type constructors that use one type T: sequence of T, set of T, bag of T and pointer to T.
- There is one type constructor that uses two types T1 and T2: array of T1 indexed by T2; T2 is then an integer, character or enumeration type.
- There are three type constructors that can use any number of types: records, unions and procedures (routines).
- Records can be recursive through the use of pointers; in almost all languages pointers can also point to items of other data types.
- A pointer to an item of type T is an independent item of type pointer to T; and operation called ‘dereference’ is needed to access the item pointed to. A reference to an item of type T is just an alias for that item; it is of type T and no operation is required to access the item.
- Routines can be a data type in some languages. They can be passed as parameters and assigned to routine variables. The only operation on them is the routine call, supplying the parameters.
- Orthogonality: two features are orthogonal if any example of the first feature can be freely and meaningfully combined with any example of the second feature.
- Restricted types: have a subset of the values of the base type. More operations are allowed on the subset that on the original set.
- Signals: asynchronous, usually with external cause. Exceptions: synchronous, with internal cause.
- Depending on the language, signals normally cause a user-provided routine to be called, smack in the

middle of everything. Exceptions can be reduced to signals, but this is inelegant.

- An exception handler is attached to some unit, and is executed instead of the remainder of the unit when an exception occurs.
- Programs consist of modules which consist of routines which consist of blocks.
- Distinguish blocks, scopes and visibility ranges.
- There is no fundamental difference between functions and operators: operators are a special case of functions and functions are a special case of operators.
- Just as we have anonymous integer values or anonymous array values, we can have anonymous routine values.
- A module is like a record in that it provides a syntactic scope the names in which can be known outside that scope.
- Specification modules: contain the information needed by user and compiler to use the items declared in the module. Implementation modules: contain the rest; are compiled once and stored in a library.
- A module *contains* a data type with its routines. An abstract data type *is* a data type and contains its routines as some of its fields.
- A generic item is a template from which one or more items can be created at compile time through instantiation. This template has parameters, typed with a generic type. A generic type is not restricted to data types and can include such things as data type names, macro names, etc.
- In Icon, an expression yields a result *and* a suspended expression; the latter is invisible to the user, but is used by the system to obtain further results if the present result causes failure of some sort.
- Icon is very suitable for search and combinatorial problems.
- Prolog uses a stack rather than local and formal variables to store and pass data.

Object-oriented languages

- Object-orientation = abstract data type + inheritance + dynamic binding.
- Abstract data type = all info on the data is included in the type: constants, variables, access methods (= procedures and functions). The variables are usually hidden form the user; access to them is through the access methods only.
- Inheritance = type extension.
- Reference semantics = the object is represented by a pointer. When the user thinks he/she passed an object, the pointer is passed.
- Overloading: a pointer to a base type object may also point to a derived type object, since the latter has (at least) all the fields of the former. So all actions on the base type can also be performed on all derived types.
- Dynamic binding: when an action is applied to a what seems a base type, the system remembers that it was actually a derived type (run-time info) and applies the action as redefined for the derived type.
- Use inheritance for "is a special kind of" relation. Use inclusion for the "has a" relation.
- Multiple inheritance means that something is both a special kind of A and a special kind of B. This is rare. Much of the need for multiple inheritance can be satisfied by generics.

Functional languages

- The program is a function from the input to the output.
- The same applies to all functions in the program → no side effects → the same input yields the same output → it does not matter when exactly you calculate the results, as long as you have the input.
- Traditionally the special data structures for functional programming are lists; this makes polymorphism easier.
- The other main data type is of course functions, which means that functions are first class data types. Functions can have functions as parameters and yield them as results; lists can contain functions.
- Polymorphism: types need not be fully determined but can have Prolog-like type variables in them.
- Currying: all functions are considered to have only one parameter and to yield a function with one parameter less. → Constants are functions with zero parameters. This allows partial parameterization and

simplifies the formalism.

- Lazy evaluation = an argument is not evaluated until its value is needed. This avoids superfluous work, prevents potential errors and is equivalent to top-down evaluation (= substitute the function first, and rather than substituting the arguments first.)
- Infinite lists: evaluate only as much as needed; this is made possible by lazy evaluation.
- I/O: consider them as infinite lists and evaluate / produce only the relevant part. This allows "I/O in languages without side effects".

Logic languages

- The domain is described as a set of condition-conclusion rules and a set of facts (= conclusions without conditions). The rules (and facts) may have logical variables in them. The actual "program" is a query asking for values of these variables that make a given "conclusion with variables" true. Such a query is called a "goal".
- The search strategy: The system looks for rules that "match" the goal, to find rules that can give conditions under which the initial goal is true. It does this by trying to match the goal to the head (= conclusion) of each rule, and if it fits, the conditions of this rule are new goals. When all goals have been reduced to facts, the system has found a way (= a set of variables) that make the original "conclusion with variables" true.
- Logical variables: can be uninstantiated (= uninitialized and the system knows it) or have a value bound to it (= point to a data structure which may contain more logical variables). Also, two uninstantiated variables can be bound together.
- Goal-head matching: a rule matches a goal when the head of the rule can be unified with the goal ("it fits""). The result of this unification is the new goal, if it exists. If it does not, the system backtracks to the latest choice point.
- Unification: A goal and a head can be unified if their names are equal and their parameters can be unified pairwise. When both parameters are data structures, they can be unified if their names are equal and their components can be unified pairwise. The is no fundamental difference between unifying a goal and a head, and unifying two data structures.
- Unification of logical variables: if one is an uninstantiated variable and the other is a data structure, the variable is bound to the data structure. If both are uninstantiated variables, they are "bound together", so that if one gets bound in a later stage, so will the other. If both are bound to values (data structures) these data structures are unified. These bindings are undone when backtracking passes through them while retreating to an earlier choice point.
- Rules describe relations, which work two (or even more) ways. A "pure" Prolog program can be run from input to all possible outputs but equally well from output to all possible inputs.
- There are several impure features in Prolog which disturb and sometimes destroy the pure relational nature of logic programming.

Parallel and distributed programming

- Parallel: multiple CPUs around shared memory. Often the processes all do essentially the same, possibly even in lock step.
- Distributed: multiple CPUs in a network. Often the processes do different things and communication times are higher.
- Parallel processes have to communicate. There are two basic methods for this: shared variables and message passing.
- Shared variables: they need synchronization measures. There is control synchronization ("wait until the data you need are there"), which waits for some other process to arrive, and mutual exclusion synchronization ("wait until the data you need are free") which waits for some other process to go away. There are locks, semaphores and monitors to do this, in order of increasing sophistication.
- Locks and semaphores: a semaphore is an integer on which the only actions allowed are "increase by 1" (V) and "decrease by 1" (P) and which can never become negative. Normally it is used to represent "the number of items available". If a P operation would make the semaphore negative if allowed to run to completion, the attempting process is suspended, and will be released as soon as somebody else does a V operation. In the long run there must be equally many Ps as Vs; this is difficult to keep straight. (A lock

is a one-bit semaphore; unlocking it twice is either an error or has no effect.)

- Monitors = a module or abstract data type that can be visited by only one process at a time. This solves the mutual exclusion but not the control synchronization. To solve the latter, processes which find they cannot continue execute a wait() on a condition variable, which puts them in a queue for that variable outside the monitor. If a process does something that might make a give condition true, it does a signal() on that condition, which alerts those in the queue for that condition. This is still difficult, but less so than semaphores.
- Message passing: The produced item (query, block of data or what not) is sent in a message to the other process. This provides control synchronization and mutual exclusion in one, and does not require shared memory, but is less efficient, and in some cases harder to use.
- The sender does a send(), the receiver may or may not do an explicit receive(). There are many choices to be made here: direct versus indirect naming of the parties involved; synchronous versus asynchronous communication; explicit versus implicit receive(); one-way versus two-way communication. The language SR has it all.
- Rendez-vous (Ada): two-way message passing with explicit receive. Remote procedure call (RPC): two-way message passing with implicit receive.
- Linda: implements a very specific shared object type, the tuple, with implicit guards and mutual exclusion. Orca: allows the user to implement their own shared object types, with explicit guards and implicit mutual exclusion.

Other Paradigms

- Paradigms can be classified according to how they access and manipulate the program state.
- Constraint programming: define the solution to your problem as a set of restrictions on the values in it. The system then has to find all the solutions that satisfy the restrictions.
- In practice the search space is too large and too badly structured for this to work.
- There are two ways out: restrict the domain to something simple; and require the programmer to structure the search space better by supplying condition-action rules.
- Access-oriented programming: changing a variable causes actions that change other variables and that propagate to do the work for you.
- Single-data-structure languages: simplify (?) programming by supplying one very powerful data type only. They are good for rapid prototyping.
- The sets of SETL provide breadth-first search, the OK/FAIL mechanism depth-first search.
- The SETL oracle function OK sets a choice (backtrack) point and returns TRUE. If a FAIL is then executed, it returns to the last choice point, clears it, and resumes the OK function, which now returns FALSE.
- Dataflow programming: each operator and function is implemented by a separate processor, which does its work as soon as the operands are available. This get you optimal fine-grained parallelism without programmer help. Also, all dataflow languages are single-assignment.
- Functional languages would be ideal dataflow languages but for one problem: they are recursive.
- First solution: do dynamic code copying when recursion is needed. This is done by tagging the variables with an incarnation tag.
- Second solution: forbid recursion, but allow iteration by other means. VAL does this by allowing a very limited form of reassignment, which creates new, tagged variables.
- All variables in Lucid are streams, which make the idea of new values without reassignment come natural.