Future Of Programming Languages

There are several pages (FutureOfSmalltalk, FutureOfJava) discussing what may come to the aforementioned languages in the future.

Here, we get to stick our fingers in the air, and engage in wild speculation (or informed prediction) as to what the future will be like for programming languages in general. (And this can include things that are decidedly non-traditional; a "programming language" can include any way that a human programs a computer, not just the sorts of things we commonly think of today...)

Each person, submit their thoughts in their own section.

The most one can expect from a GeneralPurposeProgrammingLanguage is for it to become unobtrusive. -- adapted from a quote at LambdaTheUltimate

It may be that the *most* we can hope for from the language itself (not considering community and development environment) is for it to not get in the way of our work. There is a certain amount of EssentialComplexity in our work. No matter how good or how high-level the language, we as programmers and software engineers will be left with the tasks of figuring out what to say and how to say it. And, chances are, even if we have ConstraintLogicProgramming and GoalBasedProgramming among our programming staples, we're still likely to need to spend time saying *how* or *what it means* to go about doing what we said to do. The job of the programming language is to reduce as much AccidentalComplexity as possible, leaving us with a high EssentialComplexity/(EssentialComplexity + AccidentalComplexity) ratio, and keeping this ratio high even as NonFunctionalRequirements increase.

That ratio is quite low today; there are a great many causes of AccidentalComplexity left that 'obtrude' on our programming: Violations of OnceAndOnlyOnce are forced by lack of value composition (forcing repeated parameter groups), lack of implicit context (forcing threading of context parameters... see ExplicitManagementOfImplicitContext), lack of pattern-matching (forcing repeated efforts at value decomposition, lack of modularity or extensibility (forcing efforts to be repeated in combinatorial variations for each project). We are also forced to consider details below our immediate level of concern to meet

various NFRs imposed by the environment such as resource management (file-descriptors, memory), optimizations, security, safe concurrency, realtime, etc. Often we must make irrelevant decisions, such as naming small one-off functions in languages that lack first-class functions. Sometimes we find the language obtrusive if it doesn't give us *enough* access to the low-level actions (e.g. to interface with new hardware). Many programming languages force us to write all sorts of boiler-plate code once per project to perform initialization and destruction, to interface with frameworks, etc.

Most of these problems are resolvable, but tackling them all in one language is far from trivial.

In any case, the LanguageOfTheFuture will be unobtrusive.

The following prediction is offered up for the near future (the next 10-20 years or so). Beyond that and we start getting into the realm of ScienceFiction - which isn't bad, but that's extrapolating the curve well beyond the dataset.

- The LanguageOfTheFuture will strongly borrow from both Lisp and Smalltalk; but will be neither of these (nor a direct descendant).
    - The LanguageOfTheFuture will not be C/C++, Java, or any of the DotNet languages. The CommonLanguageRuntime and the JavaVirtualMachine will likely be important targets in the near future, but not in the longer term.
- The LanguageOfTheFuture will use DynamicTyping by default; with TypeInference performed as a performance optimization (or when improved ProofOfCorrectness is needed). ManifestTyping (type annotations) will still be available, of course, to programmers who find them appropriate. In other words, it will use SoftTyping.
- The LanguageOfTheFuture will be suitable both for LargeApplications and small applications (like scripts). It will be (relatively) easy to do MetaProgramming, and write DomainSpecificLanguages.
    - Implementations of the LanguageOfTheFuture will be capable of both ahead-of-time compilation and interpretation; including the compilation of code at runtime (this capability may be elided for embedded systems and the like).
- The LanguageOfTheFuture will be OpenSource -- meaning that its specification will not be controlled by any one vendor, the specification will be published and freely available, and an OpenSource implementation will be available.

- The LanguageOfTheFuture will be OO (or actors) at the base; but borrow heavily from other paradigms (including functional).
  - *Or vice versa. It may be functional at the base, but provide powerful support for OO and Actors.*
- A theorem-proving system such as PrologLanguage will be available; and used by the language itself to assist in proving programs correct. This won't be foolproof, of course (this is still an active area of research).
  - DesignByContract (or something similar - BertrandMeyer has a trademark on the term) will be used. Preconditions, postconditions, invariants, etc. will be more than just assert statements; they will be axioms for the theorem-proving system.
- A UnitTestingFramework will be part of the language definition; each class/module/whatever will be able to specify tests that can be viewed together with the body of the code.
- Relational features (tables and records as datatypes, as well as a database type which is a collection of RelationalVariables and related constraints) will be first-class elements of the language. (Support for database access protocols, to communicate with a high-performance RDBMS, will also be provided).
- There will not be a multitude of identical types distinguished by their storage (table, array, file, stream, etc). It will simply be an attribute optimizable on the fly. The runtime can choose to store an object in memory, across the wire, or in a certain file format. All of the code that translates and map an object to a file to a table to a widget will be gone.
  - This might be simplified by decomposition to: (1) formally distinguish type from representation (allowing the runtime to control representation except where necessary, such as binary interfaces), (2) transparent distribution, (3) transparent persistence, (4) behavior/communication oriented object and service types. It is worth noting that fully dividing type (e.g. 'integer') from representation (e.g. BCD) such that programmers needn't even worry about it when interfacing with hardware and other 'external' systems (operating systems, services written in other languages) isn't entirely possible, though use of a common language for type and codec transport can serve the required roles. (See CrossToolTypeAndObjectSharing for related discussion.)
- Will have (as one mode of program entry; I do not intended to exclude other methods of specifying a program to the implementation besides typed-in or machine-generated text files) a simple syntax and grammar. Will probably be a bit

more complicated then EssExpressions (featuring some infix operators); but won't be the syntactical nightmares that are C++ and Perl. Smalltalk represents a good compromise; though many other existing languages are good examples. A parser "library" which parses the language will be part of its standard library; both to support runtime evaluation of program text, and to support cool tools like IDEs, browsers, and the like.

    - That said, there will be other ways (in the further distant future) of communicating algorithms, etc. to a computer besides text. Heck, we have this today with visual GUI-builders of all descriptions, spreadsheets, and numerous other such technologies. As things progress, more and more *will* be programmed by means other than humans typing on a keyboard.

-- ScottJohnson (with other contributors)

The language of the future will be the OS of the future. After all, the LanguageIsAnOs. It will need to support highly distributed and concurrent programming, and should support both very low level (bit manipulation) and very high level things (e.g. functors on coinductive types, artificial intelligence, object browsers, etc.)

I'm not sure all of these features in one language would be all that good an idea.(*) Things might begin to get cluttered. However, I don't like the idea of offloading everything to libraries. Maybe the LanguageOfTheFuture will have a macro system similar to Lisp's (RealMacros), and a library of standard "language extensions" that would allow you to pick and choose what you wanted. This way, the language itself could remain uncluttered, and you wouldn't need to worry about things that you didn't need. -- DaveFayram

(*) I contrarly think we can unify everything from low level to high level, and it is still feasible as a bit or a byte may well be conceptualized as objects. However, and this is where the language and the compiler have much to do, if I use objects to describe my bits, bytes, registers, etc., the resulting code should not add objects layers to the compiled code but should give a clean code that fits tight the processor language -- ColinLorrain

- I'm intentionally not making any claims/suggestions regarding the partitioning of functionality/features between "core language" and "library" in any of the above. It's a distinction important to LanguageLawyers, far less so to application programmers. (Is java.lang.String part of the Java "language" or the "library?" It gets special treatment in the language definition - a lexeme for literals - but a String in Java is just another object. Personally, for the end user, I don't think it matters.) That said, it would be a good design principle to keep the "core" as small as possible; however that shouldn't be the supreme design criterion. (Systems that try to have such

minimalist cores end up SplittingTheAtom far too often - and when that occurs their attraction as a high-level language disappears). Your mileage may vary. -- ScottJohnson

- I didn't mean to take it quite that far. :) But, I'm going to say there might still be something to this. Java strings get a special kind of hardcoded treatment that any class you write cannot get. Usually, you don't need that. Sometimes, you do. If you want a good language for munging text, then inline regexps and match operators suddenly become less a luxury and more the standard. At the same time, these things could get in the way if all you do is bit twiddling or 3d graphics. I think the LanguageOfTheFuture will allow you to extend the language, both in the classical "library" sense and in the more fundamental sense that C++ operator overloading or Lisp macros do. -- DaveFayram

Scott, it's like you read my mind. :-) -- DougMerritt

Me too! -- RobertChurch

Eventually, the language of the future will evolve beyond edit-compile-run-debug and move in a to commercial special-purpose systems like VB and LabView - programming in which the procedural and declarative code may not be in pure text form (in VbClassic, the declarative code is graphical, while in GeeLanguage of LabView, the procedural code is graphical).

As the need for hyperthreading code becomes more apparent, traditional procedural coding will become unviable because of the panic of concurrency issues. Languages that have easy, intelligent threading metaphors will be successful. Likely the compiler will handle determining where to spawn new threads and where to couple coroutines and repeated deadlocks into a single thread. Ultimately, the solution will resemble GeeLanguage, where any function that returns or passes-through more than one value is effectively creating a pseudo-thread, however the thread may not physically exist if all of the return values simply rendezvous into another function call or concurrency check. Spawning a thread will be as simple as forking a variable path. The language will support traditional text-based logic for places where it is appropriate (eg. arithmetic) which will be abused by hold-outs. High-power IDEs will become more necessary, as edit-time coupling to live systems (such as DB tables) will help automated code-maintainers keep tabs on the compatibility of the access templates to the structure. Such mapping will effectively make databases work as native members of the language, with database queries being indistinguishable from template-style function calls, and databases simply handled as objects with tables handled as inner-classes.

Top-level meta-programming concepts will be handled as local extensions to the IDE, similar to a hybrid of VB Visual Studio macros and FireFox extensions, except not executed at compile-time but instead always-on (like the .NET properties editor). These IDE extensions will be managed as part of the project, within the project's namespace.

Typing will be static, but the system will be pleasant enough that nobody will notice (as in: not Java) - as much inferencing as possible is done, with declarations or dynamic typing available where inferencing is impossible. Exception specifications will exist as auto-generated metadata instead of hand-written code, where they belong.

The language/IDE/platform will initially be implemented proprietarily, however in the interests of capturing the diverging market it will have the spec opened up and it will be cloned by opensource nerds.

Non-deterministic garbage-collection-as-panacea will fade from popularity as coders get sick of bloated "finally" blocks and inapplicability for high-performance, resource-limited, or real-time purposes. Instead, objects will be optionally instantiated into the collector, or onto the pseudo-stack, which will allow for handling similar to an auto_ptr with WeakReferences for use in the place of pointers.

Because code-maintenace/generation is handled with IDE tool assistance, the traditional subclassing of implementation will disappear, except as just one more tool. This will be replaced with a system of template-style code-generation combined with mixins, ultimately implementing the desired classes interface (and most classes will be declared as a substitutable form, in which all methods are virtual and the class need only be replaced by another class that implements the same interface, not necessarily a direct subclass).

Most filetypes will die in favour of NimbleDatabases.

And everyone will bitch about how much better Python was.

-- MartinZarate

- I'm not clear what you mean about "the declarative code is graphical" in reference to VbClassic. Do you mean that VbClassic has a form painter? This is hardly unique, innovative, or futuristic. Form painters have a long, long history that predates VbClassic, and as a GraphicalProgrammingLanguage goes, LabView is in an entirely different league from VbClassic. -- DaveVoorhis

- I suggested VbClassic because it's the only language with a form-builder that I've used. I realize that DelphiLanguage is an

older, preferable example. The relevant point is that VbClassic's form-builder is not just used to instantiate graphical widgets, but most COM objects, such as DB connections, MIDI controllers, and common-dialog-boxes. Sometimes these widgets have very complicated GUIs for configuration. That was the idea that I wanted to encapsulate - the fact that classes, mixins, static-objects, etc. will likely have their own GUIs for the developer to use to configure them. This is a stark contrast to UML, where the language is pretty much a graphical representation of a text-based C++ class definition, rather than an actual usability system. -- MartinZarate

- VbClassic is older than DelphiLanguage. Form painters existed before Windows. Form painters existed before personal computers. I agree that the ability to instantiate and configure widgets within the form painter is powerful, but that does not make VbClassic a GraphicalProgrammingLanguage. It simply means VbClassic has a nifty form painter. UML, on the other hand, *is* a VisualProgrammingLanguage, even though ExecutableUml is not yet a reality. -- DaveVoorhis

- My mistake then. I'm newer than many of the coders here, and didn't realize the age of VbClassic and form painters. AFAIK, UML does not define any procedural concepts - only class relations and methods. I agree that a domain-specific UML-like tool would likely be included within such a LanguageOfTheFuture, but I think common code-reuse concepts such as templates, mixins, external resources, etc. would be applied using object-specific GUI tools similar to those used for the more complex Vb objects (like the CommonDialogBox) - that is, a full multi-pane config tool for the object. I realize that VbClassic is hardly the epitome of such tools, I just suggested it as the LowestCommonDenominator - a common point of reference. A far better example would be the structures seen in video-game map editors such as Abuse, Qoole, or UEdit, but that would be less familiar. -- MartinZarate

- UML offers Activity Diagrams and State Transition Diagrams as a way of defining procedural "code." Carrying further your examples of Qoole and UEdit, I recall that the level editor for Duke Nukem 3D plopped you into what was essentially the normal game environment, but with the added ability to edit that environment - introduce enemies, add landscape or structural features, add powerups, etc. - while you ran around in the game world in the usual way. This was immediately intuitive and curiously exhilarating. In another limited domain, developing a spreadsheet is essentially the same idea -- software development by direct modelling and refinement, with a minimum of source code. I believe the LanguageOfTheFuture will extend this concept to general purpose software development by blending edit-time and run-time to the point that there is no longer any significant

distinction between the two, but without losing the expressive power of code. -- DaveVoorhis

- Another idea: Methods will be able to be marked pure, indicating that they will be referentially transparent, and thus cacheable, parallelizable, etc. This will allow not only enhanced (invisible to the programmer!) autothreading compiler optomizations, but also compiler support for ensuring that any particular function is safe to call in a thread.

How far into the future do you want to look?

Given enough time, I predict this:

The language of the future will be so unlike any of the languages that we know, that it's syntax will not be familiar to any of us. It will be difficult for any programmer today to identify source code in the future language, (if it will have anything that could be called source code). While actual text will likely still appear somewhere within what could be called the source code, the vast majority of the source will not be text, but something else. As a backwards compatibility feature, it might contain a tool to represent code as text, but that tool will not be able to represent *all* of the code, and will likely not be accurate on most of the code.

The language will be so different from everything we know that questions such as "is this language OO or functional" will be moot. Some activities will seem like OO programming and some like functional programming, but the similarities will be so slight that we would find it difficult to recognize them as such. The problems and the important issues related to programming languages in the future will be such that (1) people programming today will not be able to recognize them as issues and (2) we likely will not understand the differences between point of views on what the issues are.

There will be a way to check correctness and verify suitability for a purpose. But it may not (and ultimately probably will not) require anything like the writing of test cases that we do today.

Computers will likely play a much larger and much more active role in programming. Humans will likely do a whole lot less, and computers a whole lot more. We will manage to automate away most of what we already do. *Of course. But do you have any idea what sort of 'scripts' we'll be writing to accomplish this? You've written the basis for your run-of-the-mill 300 page pop-programming book, but an example, just a rough idea even, would take up a lot less space.*

AlexAusch

In future SovietRussia, source code **is** you!

Even if you are right, it is supremely unhelpful to say "the future is ineffable". How does **that** help? Pull your horizon in close enough, to the point where there **is** something to talk about.

Your final paragraph, saying that more will be automated, is less vague but also not exactly a risky prediction. More automation? Of course.

And don't be so sure of what can and can't be imagined. I've been reading ScienceFiction for decades; it has expanded the limits of my imagination considerably. :-)

-- DougMerritt

I'm not sure the future of programming languages is even a "language." I imagine that we'll be connecting components on a bus architecture. We'll rely on visualization, because the layout of the code will be much more schematic than it has been. ... -- LionKimbro

(moved to GraphicalProgrammingLanguage)

I'm going to be contrary, and say that the primary representation of programming will continue to be textual, though there may be visual views that are useful in certain circumstances or other alternate views. ... wild visualizations that may even be helpful; my point is not that no visualization progress will happen but that for serious programmers, the text representation will continue to be the "real program".

...

I'm not sure the world needs any more programming languages!

*Do I give you a hard time about **your** interests?*

So OK - maybe I just haven't fallen in love with one yet :-) -- pm

(moved to GraphicalProgrammingLanguage)

Sensafrustration: looking at the various lists of requirements, I am struck by how many of them are provided by FlowBasedProgramming: Ron's "melting pot"; connecting components on a bus architecture; GraphicalProgramming; nice simple, safe, multithreading model (so I've been told); black boxes with clean interfaces between them (data); mini-languages. You could theoretically write components using FunctionalProgramming (see http://www.jpaulmorrison.com/fbp/recurs.htm). IMO any

sequential coding technique can be used within (as) a single FBP component, so FBP adds an additional dimension, and doesn't take away any functionality. Could it be that we already have the LanguageOfTheFuture, and it's been around all this time, waiting to be noticed?! -- PaulMorrison

Of course we have the LanguageOfTheFuture already, in fact, many of them. That's what makes it so uninteresting to talk about the features of the LanguageOfTheFuture, rather than the FutureOfProgrammingLanguages. -- PanuKalliokoski

---

Two realizations have happened over the past decade or so:

1. Managed, VM-run code is good.

2. The language and the implementation should be identical.

*The first point is that `managed' (poorly defined, that), VM layers *can* be good. The second point is just wrong.*

Two fights that are and will continue to be ongoing:

1. DynamicTyping vs. StaticTyping

2. OpenSource vs. Proprietary

Interestingly, the two left-hand choices tend to correlate, as do the two right-hand ones.

Why is that?

[Possibly because StaticTyping works well in a corporate environment with lots of programmers, exactly the type of organization that can afford to pay someone to develop a language. DynamicTyping works well for one-man (or small group) hacker projects, exactly the type of people who want their languages to be OpenSource. -- JonathanTang]

SoftTyping gives a possible resolution for 1. There are some technical difficulties with doing TypeInference in a multi-language VM, but they are not insoluble. -- DavidSarahHopwood

Right now we've got two major closed-source runtime platforms targeted at StaticTyping languages (DotNet, JavaPlatform) and one OSS clone (Mono); there will be a couple of more OpenSource contenders in a few years, and the proprietary ones will continue to evolve. Languages will tend to group themselves around one VM or the other. VM holy wars will largely replace language holy wars.

[[No way: you're ignoring the power of good languages to infiltrate niches. Python's already good on the JVM (and the PSF is

financing a project to make it even better), and Jim Hugunin (who once initiated Jython, i.e., Python for the JVM) was hired last summer into Microsoft's DotNet Common Language Runtime (CLR) group on the strength of his IronPython (Python for .NET) prototype, to make DotNet even better for dynamic languages. Meanwhile, the Parrot VM is targeting Python too, not to mention PyPy. Python will keep being everywhere; often just outside the blinkered sight of management, sure, but doing its highly pragmatic job nevertheless! -- AlexMartelli]]

Languages clustered around one platform will come to resemble one another more and more: if Bloop has continuations but Floop doesn't, and they both run primarily on the Gurgle VM, it will be easier to port continuations to Floop. It may not be elegant, but it'll be ported. OOP has already become well or poorly integrated into just about every language; FunctionalProgramming idioms are following and will continue to follow.

I'd say the major innovation in terms of actual features will be increasingly refined syntax for dealing with heavily structured data: e.g. XML. It's already there, but it can be better integrated; just as dictionaries and even strings used to be considered wild and crazy datatypes, so XML-type parsing will move toward the center of basic language features.

[[Please... XML is overhyped crap! The languages of the future are already here... Lisp and SmallTalk, everything else is just a re-implementation of those.]]

That's why I said "XML-type". It's not the ideal implementation of what it does. It may or may not be the final mainstream implementation of it.

SemanticSubtyping is an interesting approach along these lines. The current implementations XDuce and CeDuce are targeted at XML, but it applies equally to other data models like EssExpressions and TermTrees. -- DavidSarahHopwood

- While I agree that LispLanguage and SmalltalkLanguage are well ahead of their time, and fine languages... I find the (rather persistent) suggestions that either of these represent the pinnacle of ComputerScience and shall never be surpassed, to be persistently obnoxious. As for "re-implementation" goes, so what? All languages are re-implementations of what came before, to some extent. Shoulders of giants and all of that. I repeat my claim above: *The LanguageOfTheFuture will strongly borrow from both Lisp and Smalltalk; but will be neither of these (nor a direct descendant).* -- ScottJohnson.

- Besides. Comparing XML to Smalltalk is apples and oranges if I've ever seen it. (Comparing XML to Lisp is too, but there is the XML-vs-EssExpressions debate, and far too many seem to

equate sexprs -- a way of encoding structured data into linear text -- with LispLanguage).

I'm going to do two lists, one for my ideal future and one for the future I think will actually come to pass:

In which reality bends to my wishes:

- The language of the future will strongly borrow from SchemeLanguage and SelfLanguage, but be neither of these.
- The language of the future will incorporate other ideas from Haskell, ML, Erlang, E, Arc, Beta, and other arcane languages.
- The language of the future will support multiple programming paradigms.
- The language of the future will be a HomoiconicLanguage supporting RealMacros.
- The language of the future will have SoftTyping.
- The language of the future will have support for RelationsAsFirstClassObjects.
- The language of the future will support AspectOrientedProgramming natively.
- The language of the future will pay more attention to flexible scoping rules.
- The language of the future will have an OpenSource reference implementation and an open specification.
- The language of the future will come with a powerful & pretty IDE.
- The language of the future will interoperate seamlessly with existing languages.

All of these points except the last are compatible with each other. I'm unconvinced that being able to interoperate seamlessly with some existing languages, such as C or C++, is compatible with the rest. To do that you might have to support only restricted dialects of C/C++, and it is not clear that this would be accepted by most C/C++ programmers, or that it is worth the effort. -- *DavidSarahHopwood*

In which I face the grim facts:

- The language of the future will be an incremental evolution of existing programming languages (likely Java, but probably with bits of Perl, Python, C++, and C# thrown in).
- The language of the future will have StaticTyping, but with better support for ParametricPolymorphism and GenericProgramming than existing type systems.
- The language of the future will not have any sort of macros or extension mechanism.
- The language of the future will be ObjectOriented, but may incorporate some ObjectFunctionalPatterns.

- The language of the future will be based on something OpenSource, but will be taken over and bastardized by a major corporation.
- The language of the future will come with a powerful & pretty IDE.
- The language of the future will interoperate seamlessly with Java and C. It won't interoperate with anything else (except possibly DotNet, if MicroSoft is the bastardizing corporation).

-- JonathanTang

We shall all be using the Language Of The Future by 2075, because by then Perl 6 will have been released. -- EarleMartin

- Don't count on ArcLanguage by then, though.
- Whatever language *does* dominate in 2075, there will be a buncha folks claiming that Smalltalk-80 and/or CommonLisp circa 1989 are still better.
- *And the sad thing is that they'll probably be right! :)*

"I don't know what the language of the year 2000 will look like, but I know it will be called Fortran." -- *Tony Hoare [CarHoare], apparently on a card distributed during the 1982 AFIPS National Computing Conference.*

But if there was any single "language of the year 2000", it wasn't called Fortran.

- Some made the same critique about his statement in regard to what was in use in 1982, also. Presumably his point is that Fortran will never die, it just keeps getting features added to it from other languages, which so far is still true.
  - *Hoare's statement could equally well be applied to Lisp. Unlike Fortran, however, it's not that Lisp keeps getting features added to it from other languages; rather, the other languages keep getting Lisp features added to them. (InSovietRussia, Lisp adds features to you!) -- JosephDale*
  - Any sufficiently complex, ad-hoc, informally-specified, bug-ridden, slow^H^H^H^Hprematurely optimized system is someone's Fortran implementation? -- JonathanTang, with apologies to PhilipGreenspun

---

Simple: The language of the future will have all the HolyWar features I like, and banish those that I don't. -- FillInTheBlankWeenie

---

In the future, we'll take RichText and extend the concept to RichSource *(see IntentionalProgramming)*. You'll be able to freely

mix the source code, comments, graphics, and the Graphical UI parts of the program in one non-ASCII document.

Why non-ASCII? Wouldn't an XML application suffice for such a thing? While I wouldn't want to hand-edit such a beast on a regular basis; there are many reasons that PowerOfPlainText should be considered.

Or, we could use EssExpressions rather than XML (since XmlIsaPoorCopyOfEssExpressions).

- XML doesn't do Layers and Markup, neither does HTML. A markup language which allows the creation of annotation, is a very necessary part of my vision in this picture. Yes, I know about the utility of plain text, but there is always something traded away. I'm tired of the trade.
- *What do you mean by "layers" and "markup", and why do you think XML can't do them? What do you think the "M" in XML stands for, anyway? And what does binary-vs-text have to do with "layers" and "markup", as well? The big advantage that binary has over text-based file formats is that it's easier with binary to develop schemas with fixed record lengths (though one can do that in text with padding), allowing files to be updated-in-place rather than read and rewritten. For an application like a database, this can be a big performance win (which is one reason that XmlDatabases are usually a bad idea). For a compiler; I suspect that doesn't matter much.*
  - Markup - Source code is one layer, notes about the source code could be used to mark a span of the text. A second note could annotate part of that span and the code that follows. This requires overlapping tags, which are structurally prohibited from XML, and actively discouraged in HTML.
    - No, it doesn't require forbidden overlapping tags. You can represent anything you like in XML. E.g. <ANNOTATE flag="on"/>. The syntactic requirements for XML are very reasonable, and after all result from cleaning up SGML based on experience with the latter's problems. The syntax of XML need not be one to one with the semantics of the XML tags. One can semantically represent circular structures in XML, for instance.
  - Layers - One should be able to select layers of markup to be viewed, just like layers in PhotoShop. Comments could be one of those layers, the symbol table links could be another, and the generated object code could be yet another layer.

You'll then work with a BidirectionalCompiler, which will allow you to view and edit the symbol table, resulting in automatic updates to the source. (In essence, a RefactoringBrowser on steroids) It'll make debugging a bit easier too. -- MikeWarot

[Finding the existing terminology to fit my internal GrandVision is proving to be much tougher than I thought]

The language of the future will be like a chemist's version of AssemblyLanguage, because we'll mainly be programming nanomachines, and the available resources and sophistication of the CPU will once again be sparse. Farther in the future, we'll figure out new and entirely different high-level languages to simplify the task.

Without looking (much) at the other entries:

Short term:

- Dynamic features in languages will become more and more prominent.
- There will be an outburst of "weird" languages in server-side program development.
- There will be a few attempts to invent new "something-oriented" ProgrammingParadigms.
- Side-effect purity will become more mainstream.
- There will be a lot of OpenSource languages that are nifty amalgamations of features, and in ten years, some language designed by a single mind that makes a real paradigm shift.
- Parallelism, distribution and security will become more and more integrated to core languages.
- Big language wars wane, and the programmer community will be divided to more subcultures, who emphasize the pros of their own LanguageOfChoice.
- People will return to GOFAI techniques (but in a new form; XML, maybe).
- More and more programs will ship with an integrated debugger (for in-production debugging).
- Every tool that has been around "long enough" to gain popularity will have some kind of built-in command language.

Long term:

- The need of intercommunication will give rise to designing a unified information model for all programs.
- This unified information model will have libraries for NLP, NLG, and possibly inference for a multitude of languages.
- Approaches that give a clean programmable interface to specialized hardware for neural networks, quantum computation, and the like, will slowly gain ground.
- There will be an increasing amount to make programming more tangible to non-programmers: drag-and-drop programming, interactive programming, IDE's not presupposing programming experience (like DrScheme),

transparent GUI's, and languages that have "beginner syntax" and "expert syntax" separately.

Having now read some of the other entries:

- The future of programming languages is not a language.
- There will be no single LanguageOfTheFuture; rather, corporations, hackers, and PracticalProgrammers will evolve their own.

-- PanuKalliokoski

The things which a future programming language could provide -

1. Semantic clarity
2. Database Independence
3. User Interface Independence
4. Communication Interface Independence
5. Evolutionary Development - needs based - when you need it, find it and use it, or build it.
6. Introspection - the ability to produce alternate views of 'source code' - the ability to extract relevant information from the code.
7. Ecumenism - allowing pragmatic inclusion of other languages.

I believe that fulfilling these requirements means that whatever form the future languages take, it will be simpler than now.

And the first element in a list of items will be referred to as position 1, not offset 0.

Blech.
<*http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html*>

And a comparison will return true when the first and only element of an array is equal to a scalar element.

```
{"Spade"} = "Spade"
{""} = ""              ; * An empty array is equal to null
```

-- PeterLynch

- *Please, please, don't do that. That can destroy tools that reason about correctness of code. Different types are different types, and automatically coercing them on comparison is a likely cause of unexpected bugs.*

I expect that type checking will be moved from compilers in the future. But that is only my opinion. The current paradigm is still that the type checking at compile time is beneficial, so from that perspective, OK, I see why you would object to comparing apples

to oranges. But these are Tools, in the dataset named "Tools", and "Spade" is one of them. When I compare a bag containing a Spade to a Spade, in terms of Tools, I have equality. A Spade in one hand, and a Spade in the other.

- *I disagree with that fundamentally, as would every mathematician. You have a spade in one hand and a bag in the other. Confusing zero with the set containing zero, or one with the set containing one, is a classic undergraduate error that for some people spells the end of any hope of progressing in mathematics. I most sincerely hope it doesn't get promoted by computing. Consider, is there much difference between a lion in a cage, and a lion?*
  - lion 1 has a location of "in cage", and lion 2 has a location of "outside cage". They are both lions.
- *Yes, they are both lions. But you are suggesting that [1]==1, or ["Spade"]=="Spade", and by analogy, 'Lion in cage'=='Lion'. Yes, the lion that is in the cage may be equal to the lion that is not in the cage, but Lion_in_cage is fundamentally different - I'd feel a lot happier being near it, for example. It may be my failing, but a set with one element is not the same as, and should not compare equal to, the element not in the set. A list with one element is not the same as, and should not compare equal to, the element not in the list. A bag with an angry cat is not the same as an angry cat - being in a bag makes a very real semantic difference.*

The examples above are more about Singularity/Plurality than Datatyped or not Datatyped. In a future language, I can imagine atoms becoming arrays when the second value is entered. Plurality, like lists, will be a natural consequence of the operations performed.

- *I disagree. You are advocating the deliberate confusion of a set with an element. That is an issue of type, no matter where it is checked, or not checked. Is a list containing an empty list different from an empty list? You are suggesting not. Can open - worms **everywhere**.*

Because I have used MultiValued languages, I see this differently. Plurality and Singularity are separate from datatypes. Any Datatype can be stored in singular or plural form. An item becomes plural when it contains more than one element. It may be a member of a plural data base column, but if the item within the instance has only one value, and that value is the first value, then it is an atomic value.

```
a = "Dog"
b = ""      ; * Initialize b to the null string
b<1> = "Dog"  ; * Set the first element of b to "Dog"
Print a
```

"Dog"

```
Print b<1>
```

"Dog"

```
Print b
```

"Dog"

```
Print a = b
```

True

```
Print b<2>
```

"" (Null)

- *I have no idea what language this is in, but I wouldn't touch it with a barge-pole. I use Python extensively which has lists, tuples and dictionaries (and sets in newer versions) and in each case there can be none, one or more elements. In each case there is a semantic difference between a collection with exactly one element and that element. I most strongly believe that's how it should be.*
- *Why do you want [[]] == []?*

That's PickBasic, son! It's been "usefully" dead for quite a few years now.

- No. Well yes, but no. OpenInsight from Revelation Software is a grandchild of PickLanguage, and this would work in that language. It would not work in Dick Pick's language, though the concepts are the same. OpenInsight is not dead - it is slightly more alive than it was 20 years ago.

I think this problem is to do with semantics. It is not possible to compare an apple with a banana, but conventional languages accept that you may want to. Typecast is the mechanism.

```
I read the statement -
   If a = b
as "If the contents of a are the same as the contents of b".
The above example is consistent with that view.
```

There is something I don't understand about this argument. You're talking about the fundamental difference there is (or there is not) between a lion and a lion in a cage. Great, sounds funny. But [Lion] is NOT a lion in a cage; it is a cage that happens to contain a lion. It could contain something else, or nothing. You can feed a lion, you can open a cage, but I wouldn't recommend to open a lion, and feeding a cage could be an interesting experience. In short, I can imagine a call to [Lion].open(), I can imagine a call to Lion.feed(), but Lion.open() is just plain nonsense! -- PhilippeDetournay

Other than a vet, I see no reason to want to open a lion. Whatever programmer tried to open a lion should have been ignored (open could do nothing), or maybe fired.

AIUI, this is inspired by the treatment APL, Matlab, *Lisp, and similar auto-parallelising languages use - although I think it takes the idea a little too far, as you want to be able to operate on the collection as a whole (or a set of dimensions from it, since the LanguageOfTheFuture will have native multidimensional collections, I think, at least for array-like collections), and the runtime need to be able to tell whether you're looking at the object itself or a reference for the sake of core functions (especially reduce... functions). I'm a strong believer in exposing the runtime to the application (or, in practice, to the libraries, perhaps like LaTeX does it with @-commands but more elegantly).

I suggest that in the future no human will write code. It will be done by the machine. AS we move onwards and compilers become better and allow for more abstraction and as we build more and more intelligence into the compilers you will achieve the StarTrek "Computer - show me the status of the ship". The computer will then work out what you mean and deliver the information. Programmers will no longer be required as programs will write programs.

But see TeachMeToSmoke.

I was going to say the same thing, well I didn't hope for such help as StarTrek computer, because that also "thinks", creates holograms that can come out of the special room etc, but at least I would like to say "make a loop here" or even better "extract this interface and make an abstract base class with this and that", a sort of "junior developer" that can implement most of what you say you would like to without hating you. -- MicheleVivoda

---

It is amazing how close this page comes to the ideas of LanguageOrientedProgramming as described in http://www.onboard.jetbrains.com/articles/04/10/lop/.

I see a possibility for macro contraction, or other forms of idiom translation. People love and/or hate macros because you can end up programming in your own custom language. What if when I got a piece of code from someone else, I could expand all the macros that were somehow designated as part of the author's personal set, rather than specific to the problem domain. Then I run a macro contraction tool to replace matching code with *my* personal macros. Probably bored compiler optimizers would be the best candidates for writing this tool.

You know, there might be the germ of a great idea here - I am starting to get fed up with all the different ways of saying, e.g. while (for, do while), break vs. leave, etc. After a bit, all the languages run together in my head! So, if we could come up with

an Esperanto of programming languages, everyone could translate them into their favorite syntactic sugar! I know, we'd still have to agree on the Esperanto... Oh well...

That sounds very much like Syn*:*

- <http://www.interstice.com/journals/Simon/20041021.1.h.html>

I hope for some SeparateMeaningFromPresentation. Thus whether a language uses semi-colons or not or uses curly-braces instead of end-x syntax may be a personal choice such that the view can be altered for personal preference. -- top

---

This topic has also been talked about at LambdaTheUltimate:

```
http://lambda-the-ultimate.org/node/687
```

Of interest is in particular the comment by FrankAtanassow: SomeWordsOfAdviceOnLanguageDesign.

Repeated interest in JulyZeroFive

See also PerfectLanguage, IssuesForLanguageDesigners, QuestForThePerfectLanguage, ProgrammingParadigm, LanguageTrends, LetsDesignProgrammingLanguage, IdealProgrammingLanguage

---

CategoryProgrammingLanguage

Last edit November 8, 2014, See github about remodeling.