Chapter 1 : PL/I - Wikipedia

PL/I (Programming Language One, pronounced / p iË• É› l w ÊŒ n /) is a procedural, imperative computer programming language designed for scientific, engineering, business and system programming uses. It has been used by various academic, commercial and industrial organizations since it was introduced in the s, and continues to be actively used.

An inline FILE sort is provided by the operation: Two instructions do it all. Popup windows, borders and full color control are all built in. Consider this "popup" routine: Column 3, Line 2 "Hi there! Next line "Tap any key. Programs can be coded with a traditional text editor but a visual Integrated Development Environment IDE is also provided. The IDE incorporates project management, forms design, language specialized editor, debugging tools, and related tools. The forms designer allows screens to be be created and manipulated with standard visual tools. Properties as well as code segments are attached to each screen object then the objects are packaged into a shell program which provides the overall structure for the application. The GUI implementation is conceptually similiar to other object oriented languages. Objects are defined and given properties. The application program activates various objects then waits for an event. The text based code is evaluated and the interpreter faithfully replicates keyboard and screen handling within what is essentially a windows text box. Interestingly, both GUI objects and texb based components can be combined in the same program and same screen, which can be very useful. The Server program can run on a Windows, Unix or Linux platform. The application program is executed on the server. The client program on the remote workstation handles the user interface. The client program can be downloaded or run from the server via a standard internet browser. A web page would provide a link to the client program on the server. When clicked, the client is downloaded to the remote machine and begins to communicate with the server. All file processing is centralized at the File Manager machine thus optomizing file operations. Mid Michigan Computer Consultants, Inc.

Chapter 2 : The XPL Programming Language

PL/I is a legacy programming language invented by IBM in the early s. The intention was to create a language which would have the scientific computing capabilities of Fortran, and the business data-processing power of COBOL. A Quick Note on the Name. Many people prefer to call this programming language PL/1.

Taken by every sophomore in our program, it has evolved into a pretty interesting course, and I find myself talking about it with various people I meet. I apologize in advance for any errors I have made, e. Overview CMSC aims to give students a broader view of computation and programming languages. We want to introduce them to new ways to think about and solve programming problems, exemplified by different languages. We also try to get them to think carefully and rigorously about what languages mean and how they are implemented , which should help them use those languages more effectively. During , we teach them functional programming using Objective Caml , scripting languages using Ruby , and safe low-level programming using Rust. We also teach operational semantics and lambda calculus as a means to formally understand languages and computation. Finally, we teach some core elements of secure coding, a task that many of students are sure to undertake, whether they realize it at the time or not. We also present some of the history of programming languages. Scripting with Ruby The course starts by introducing Ruby. We immediately consider the tradeoff of this design to the more familiar design of explicitly and statically typed declarations as in C and Java. Its support for syntactically lightweight regular expressions, arrays and hashes, simple pattern matching, and other features make it well suited to text processing. Students thus see the power of domain-specific constructs in making certain tasks easy. Their project during this part of the class involves building simple data structures and classes, and using regular expressions, to do some text processing. The experience of designing regular expressions to match structured text in these early projects makes it easier for us to later delve into how regular expressions work, along with their limitations. That said, we did have technical reasons for choosing Ruby. The result is the object representing 3. I also like that Ruby has long had support for higher-order programming, in the forms of code blocks and procs. Learning code blocks smooths the transition to full higher-order programming in OCaml. For example, in Ruby I could write [1,2,3]. This is like List. Many idiomatic Ruby programming patterns use code blocks, so by the time we get to full closures in OCaml the basic concepts are pretty familiar. Later in the course we come back to Ruby as a web application-authoring language, which elevates security concerns. Functional languages make this programming style more natural. Two key reasons for this are the presence of easy-to-write first-class functions i. As one explanation for why one might teach FP, Swarat Chaudhuri wrote on this blog some time ago, There are good reasons [to teach] functional programming. In imperative languages like C, a line of code within a procedure can affect the entire global state of the program. In contrast, in functional languages, such side effects are severely limited. This makes it much easier to reason about correctness and security: Researchers have also frequently argued that functional program design makes it fundamentally easier to exploit parallelism. I like to point out that both features and core design elements of mainstream not primarily functional programming languages had their origins in functional languages. As such, learning a modern functional language can give you a leg up on the next big language, or on future features of your current favorite. Garbage collection was part of LISP , in Lambda terms anonymous, first-class functions , implemented as closures also come from LISP and are a part of most modern languages today, e. Generic types started out as parametric polymorphism in ML in the late s, and earlier in more theoretical settings, e. Type inference is now part of Java, C , Rust, and many other languages. Looking to the future, I think that pattern matching e. It is a key part of Rust, and it is coming to a new version of Java. As a last motivation, I also mention that notable companies have taken the plunge and use functional languages like OCaml, Haskell, Clojure, Erlang, and Scala directly in major projects. Like many of the schools that teach FP, we use OCaml. And OCaml is easier to pick up starting from programming languages the students have already seen; e. Like Ruby, OCaml programs are relatively terse and require no type annotations. These similarities help us perform an initial transition, but immediately we start to see the differences, e. This style is useful in and of itself, and it also lays the foundation for talking about context-free grammars, parsing, and operational semantics in more depth later. Functional programming is the largest unit in , covering 3. Impact and Alternatives Despite all of the attention, not many students are persuaded that OCaml is the best thing, ever. I took a poll on the last day of class last semester, asking which programming language the students liked best. At least OCaml was preferred to C! One interesting suggestion that was made to me was to use Scala rather than OCaml. I may say more about why in a future post.

Chapter 3 : Programming Language Definition | Webopedia

PL/1 was used in Multics as a system programming language, one of the first attempt to write OS in high level language (that attempt inspired Unix and C has by-and-large PL/1 semantic and structure). PL/1 compilers used to represent to edge of compiler development with IBM optimizing and debugging compliers for many years representing the top of the class.

These keywords divide the block into a declarative part, an executable part, and an exception-handling part. The declaration section is optional and may be used to define and initialize constants and variables. If a variable is not initialized then it defaults to NULL value. The optional exception-handling part is used to handle run time errors. Only the executable part is required. A block can have a label. Blocks can be nested â€” i. The interactive tool or program runs the block once. The block is not stored in the database, and for that reason, it is called an anonymous block even if it has a label. This returned value may be a single scalar value such as a number, date or character string or a single collection such as a nested table or varray. User-defined functions supplement the built-in functions provided by Oracle Corporation. The only out value from the function should be the value it returns. Procedure[edit] Procedures resemble functions in that they are named program units that can be invoked repeatedly. The primary difference is that functions can be used in a SQL statement whereas procedures cannot. Another difference is that the procedure can return multiple values whereas a function should only return a single value. The procedure begins with a mandatory heading part to hold the procedure name and optionally the procedure parameter list. A simple procedure might look like this: The standalone or package procedures, stored in the database, are referred to as " stored procedures ". Procedures can have three types of parameters: An IN parameter is used as input only. An IN parameter is passed by reference, though it can be changed by the inactive program. The program assigns the parameter a value and that value is returned to the calling program. That initial value may or may not be modified by the called program. Any changes made to the parameter are returned to the calling program by default by copying but - with the NOCOPY hint - may be passed by reference. The use of packages promotes re-use of code. Packages are composed of the package specification and an optional package body. The specification is the interface to the application; it declares the types, variables, constants, exceptions, cursors, and subprograms available. The body fully defines cursors and subprograms, and so implements the specification. Two advantages of packages are: They support object-oriented programming features like function overloading and encapsulation. Using package variables one can declare session level scoped variables, since variables declared in the package specification have a session scope. Database trigger A database trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes itâ€”that is, the trigger firesâ€—whenever its triggering event occurs. While a trigger is disabled, it does not fire. You specify the triggering event in terms of triggering statements, and the item they act on. The trigger is said to be created on or defined on the itemâ€—which is either a table, a view, a schema, or the database. You also specify the timing point, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects. If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a DML trigger. If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a system trigger. Purpose of triggers[edit] Triggers can be written for the following purposes: Generating some derived column values automatically Enforcing referential integrity Event logging and storing information on table access Auditing.

Chapter 4 : PL/B - Programming Language for Business

PL/I programming restrictions and requirements There are some restrictions and requirements for a PL/I program that is to be used as a CICS application program. Language Environment coding requirements for PL/I applications.

The IBM extensions are summarised in the Implementation sub-section for the compiler later. Although there are some extensions common to these compilers the lack of a current standard means that compatibility is not guaranteed. A subset of the GY [16] document was offered to the joint effort by IBM and became the base document for standardization. The major features omitted from the base document were multitasking and the attributes for program optimization e. Proposals to change the base document were voted upon by both committees. In the event that the committees disagreed, the chairs, initially Michael Marcotty of General Motors and C. Hoare representing ICL had to resolve the disagreement. Further development of the language occurred in the standards bodies, with continuing improvements in structured programming and internal consistency, and with the omission of the more obscure or contentious features. Discussion of a single item might appear in multiple places which might or might not agree. It was difficult to determine if there were omissions as well as inconsistencies. Andrews of IBM undertook to rewrite the entire document, each producing one or more complete chapters. It was the first, and possibly the only, programming language standard to be written as a semi-formal definition. To fit a large compiler into the 44 kilobytes of memory available on a kilobyte machine, the compiler consists of a control phase and a large number of compiler phases approaching The phases are brought into memory from disk, one at a time, to handle particular language features and aspects of compilation. Each phase makes a single pass over the partially-compiled program, usually held in memory. Reflecting the underlying operating system, it lacks dynamic storage allocation and the controlled storage class. Unlike the F compiler, it has to perform compile time evaluation of constant expressions using the run-time library, reducing the maximum memory for a compiler phase to 28 kilobytes. Macros were defined to automate common compiler services and to shield the compiler writers from the task of managing real-mode storage, allowing the compiler to be moved easily to other memory models. The gamut of program optimization techniques developed for the contemporary IBM Fortran H compiler were deployed: This compiler went through many versions covering all mainframe operating systems including the operating systems of the Japanese PCMs. The team was led by Brian Marks. This format is interpreted by the Checkout compiler at run-time, detecting virtually all types of errors. Pointers are represented in 16 bytes, containing the target address and a description of the referenced item, thus permitting "bad" pointer use to be diagnosed. In a conversational environment when an error is detected, control is passed to the user who can inspect any variables, introduce debugging statements and edit the source program. Over time the debugging capability of mainframe programming environments developed most of the functions offered by this compiler and it was withdrawn in the s? UniPrise Systems , Inc. Additional data types and attributes corresponding to common PC data types e. Improvements in readability of programs â€” often rendering implied usages explicit e. The ordinal facilities are like those in Pascal , e. Competitiveness on PC and with C[edit] These attributes were added: The DATE pattern attribute for controlling date representations and additions to bring time and date to best current practice. Compound assignment operators a la C e. Additional parameter descriptors and attributes were added for omitted arguments and variable length argument lists. Program readability â€” making intentions explicit[edit] The VALUE attribute declares an identifier as a constant derived from a specific literal value or restricted expression. The package construct consisting of a set of procedures and declarations for use as a unit. It was heavily used by Daisy Systems for electronic design automation software on the "Logician" family of special-purpose workstations. It has been widely used in business data processing [49] and for system use for writing operating systems on certain platforms. The pioneering online airline reservation system Sabre was originally written for the IBM in assembler. It remained a minority but significant player. First, the nature of the mainframe software environment changed. On mainframes there were substantial business issues at stake too. Compiler development was expensive, and the IBM compiler groups had an in-built competitive advantage. Many IBM users wished to avoid being locked into proprietary solutions. But a number of features of significance in the early implementations were not in the Standard; and some were offered by non-IBM compilers. And the de facto language continued to grow after the standard, ultimately driven by developments on the Personal Computer. IBM has continued to add preprocessor features to its compilers. The preprocessor treats the written source program as a sequence of tokens, copying them to an output source file or acting on them. Tokens are added to the output stream if they do not require action e. Subsequent occurrences of PI would be replaced by 3. The structure statements are:

Chapter 5 : PL/B language document

INTRODUCTION PL/I is a large and powerful multipurpose programming language. The intent of the designers of PL/I was to create.

The first two posts covered programming language styles and mathematical concepts. This course topic might strike you as odd: Why teach security in a programming languages course? I believe that if we are to solve our security problems, then we must build software with security in mind right from the start. To do that, all programmers need to know something about security, not just a handful of specialists. Security vulnerabilities are both enabled and prevented by various language mis features, and programming anti patterns. As such, it makes sense to introduce these concepts in a programming languages course, especially one that all students must take. This post is broken into three parts: Security is a programming languages concern The Status Quo: Too Much Post-hoc Security There is a lot of interest these days in securing computer systems. This interest follows from the highly publicized roll call of serious data breaches, denial of service attacks, and system hijacks. In response, security companies are proliferating, selling computerized forms of spies, firewalls, and guard towers. It might be that these efforts are worth their collective cost, but call me skeptical. I believe that a disproportionate portion of our efforts focuses on adding security to a system after it has been built. Is your server vulnerable to attack? If so, no problem: Prop an intrusion detection system in front of it to identify and neuter network packets attempting to exploit the vulnerability. As computer security experts have been saying since at least the 60s, if you want a system to actually be secure then it must be designed and built with security in mind. Waiting until the system is deployed is too late. Building Security In There is a mounting body of work that supports building secure systems from the outset. These are good efforts. For this shift to happen, we need better education. Security in a Programming Class Choosing performance over security Programming courses typically focus on how to use particular languages to solve problems efficiently. Functionality is obviously paramount, with performance an important secondary concern. If you argue that security is not important for every application, I would say the same is true of performance. Indeed the rise of slow, easy-to-use scripting languages is a testament to that. But sometimes performance is very important, or becomes so later, and the same is true of security. Indeed, many security bugs arise because code originally written for a benign setting ends up in a security-sensitive one. As such, I believe educators should regularly talk about how to make code more secure just as we regularly talk about how to make it more efficient. To do this requires a change in mindset. A reasonable approach, when focusing on correctness and efficiency, is to aim for code that works under expected conditions. But expected use is not good enough for security: Code must be secure under all operating conditions. Normal users are not going to input weirdly formatted files to to PDF viewers. As such, students need to understand how a bug in a program can be turned into a security vulnerability, and how to stop it from happening. Our two lectures in CS on security shift between illustrating a kind of security vulnerability, identifying the conditions that make that vulnerability possible, and developing a defense that eliminates those conditions. For the latter we focus on language properties e. Security Bugs In our first lecture , we start by introducing the high-level idea of a buffer overflow vulnerability, in which an input is larger than the buffer designed to hold it. We hint at how to exploit it by smashing the stack. Once again, these vulnerabilities all have the attribute that untrusted data provided by an attacker can be cleverly crafted to trick a vulnerable application to treat that data as code. This code can be used to hijack the program, steal secrets, or corrupt important information. Coding Defenses It turns out the defense against many of these vulnerabilities is the same, at a high level: We should make sure an input is not larger than the buffer allocated to hold it, so the buffer is not overrun. For the other four attacks, the vulnerable application uses the attacker input when piecing together another program. For example, an application might expect user inputs to correspond to a username and password, splicing these inputs into a template SQL program with which it queries a database. But the inputs could contain SQL commands that cause the query to do something different than intended. The same is true when constructing shell commands command injection , or Javascript and HTML programs cross-site scripting. The defense is also the same, at a high level: None of this stuff is new, of course. Most security courses talk about these topics. Our security project reflects the defensive-minded orientation of the material. While security courses tend to focus on vulnerability exploitation, CS focuses on fixing the bugs that make an application vulnerable. We do this by giving the students a web application, written in Ruby, with several vulnerabilities in it. Students must fix the vulnerabilities without breaking the core functionality. We test the fixes automatically by having our auto-grading system test functionality and exploitability. Several hidden tests exploit the initially present vulnerabilities. Through exploitation, the attacker is able to execute code of their choice on the target system. Use-after-free errors, double-frees, and buffer overflows are all examples. Programs written in type-safe languages, such as Java or Ruby, 1 are immune to these sorts of memory errors. Writing applications in these languages would thus eliminate a large category of vulnerabilities straightaway. And we see that these systems are regularly and increasingly under attack. What are we to do? Type safety without GC In , the Mozilla corporation which brings you Firefox officially began an ambitious project to develop a safe language suitable for writing high-performance programs. The result is Rust. In Rust, type safety is possible without garbage collection, which is not true of any other mainstream language. Ownership aka linear typing is used to carefully track pointer aliasing, so that memory modified via one alias cannot mistakenly corrupt an invariant assumed by another. Lifetimes track the scope in which pointed-to memory is live, so that it is freed automatically, but no sooner than is safe. These features support managing memory without GC. We provide a simple programming project to familiarize students with the basic and advanced features of Rust. Assessment I enjoyed learning Rust in preparation for teaching it. I had been wanting to learn it since my interview with Aaron Turon some years back. The Rust documentation is first-rate, so that really helped. I also enjoyed seeing connections to my own prior research on the Cyclone programming language. Rust has taken these ideas much further, e. A big challenge in Cyclone was finding a way to program with unique pointers without tearing your hair out. Nevertheless, Rust is a much-loved programming language , so the language designers are clearly doing something right! Oftentimes facility is a matter of comfort, and comfort is a matter of education and experience. As such, I think Rust fits into the philosophy of CS , which aims to introduce new language concepts that are interesting in and of themselves, and may yet have expanded future relevance. Conclusions We must build software with security in mind from the start. Educating all future programmers about security is an important step toward increasing the security mindset. In CS we illustrate common vulnerability classes and how they can be defended against by the language e. We might also interest them to learn more about security in a subsequent security class. In writing this post, I realize we could do more to illustrate how type abstraction can help with security. Ruby is dynamically typed, but arguably type-safe. Old habits, and legacy code, die hard.

Chapter 6 : PL/SQL Tutorial

Programming language I (PL/I) is both a procedural and an imperative programming language that was designed for engineering, scientific and systems programming, and business applications. It has been primarily used by the academic, industrial and commercial sectors since its introduction in the s and is still in active use as of

Here is relevant quote form Wikipedia: IBM took NPL as a starting point and completed the design to a level that the first compiler could be written: These manuals were used by the Multics group and other early implementers. The first compiler was delivered in In was slow multipass compiler that was designed to operate with tiny memory amount of memory 64K that was available in first mainframes. Now it is difficult to comprehend how almost the whole computer science was hijecked by this primitive religious doctrine. But it was true and has demostrated quite well that the virtues ascribed to academic scientists are way overblown. Heretics who are ready to be burned on the stake defending the truth are as rare in academic community as among commoners. May be even more rare. Complete corruption of academic economics, conversion of the majority of them into intellectual defenders of interests of financial oligarchy that we observed since say this pretty load and clear. And computer scientists are not that different those days. They also depends on grants and want tenure at all costs. Which requires certain compromises. The idea was "simple, attractive and wrong" and due to this it soon became more fashionable that drunks fights in pubs in England. It attracted huge amount of verification snake oil salesmen and "computer science crooks" who often flooded computer science departments and eliminated useful research. Dijkstra played the role of a fervent Ayatollah. This controversial character has the religious zeal of Luther, but was more misguided. But cutting the language to the subset suitable for system programming they cut way too much. Usage round brackets as delimiters for conditional expressions in if statements is another. Just those two errors cost C programmers immense amount of lost hours in trying to find errors that should not exist in properly designed language in the first place. Later object orientation OO became fashionable with its own crowd of snake oil salesmen. Each of those splashes generated huge crowd of crooks and religious fanatics, as well as promoted corruption in computer science departments. As for computer science departments, the level of corruption from early 80th became probably pretty close to corruption of economic professions with its bunch of highly paid intellectual prostitutes or outright criminals masquerading as professors. To fit a large compiler into the 44kByte memory allowed on a 64kByte machine, the compiler consisted of a control phase and a large number of compiler phases approaching The phases were brought into memory from disk, and released, one at a time to handle particular language features and aspects of compilation. The level of diagnostics in F-compiler was good although not perfect. Reflecting the underlying operating system it lacked dynamic storage allocation and the controlled storage class. They are still unsurpassed pair of compilers for a very complex language despite being 40 years old. Unlike the F compiler it had to perform compile time evaluation of constant expressions using the run-time library - reducing the maximum memory for a compiler phase to 28 kilobytes. Macros were defined to automate common compiler services and to shield the compiler writers from the task of managing real-mode storage - allowing the compiler to be moved easily to other memory models. The gamut of program optimization techniques developed for the contemporary IBM Fortran H compiler were deployed: This compiler went through many versions covering all mainframe operating systems including the operating systems of the Japanese PCMs. The team was led by Brian Marks. This format was interpreted by the Checkout compiler at run-time, detecting virtually all types of errors. Pointers were represented in 16 bytes, containing the target address and a description of the referenced item, thus permitting "bad" pointer use to be diagnosed. In a conversational environment when an error was detected, control was passed to the user who could inspect any variables, introduce debugging statements and edit the source program. Over time the debugging capability of mainframe programming environments developed most of the functions offered by this compiler and it was withdrawn in the s? This was a good, old IBM with strong engineering core, before the current wave of outsourcing destroyed the company engineering culture. The real masterpieces of software engineering Incredible achievement of IBM engineering talent. Like System hardware and assembler language they stood as monuments to "good old IBM". It was often called Multix, which is formally incorrect but a shorter name. Although Multics was much derided at the time by its critics, history has shown that it was a real pioneer on OS design which introduced concepts now taken for granted. Many ideas implemented in Multics were years ahead of their time. Key ideas see multix were extremely difficult to implement on primitive hardware that existed at mid 60th, so the fact that they materialized is nothing but amazing. Multics began as a research project and was an important influence on operating system development. It also pioneered the use of high-level languages for writing operating systems. It also was one of the first OS which paid serious attention o the security. There has been some discussion of ways in which the operating system might be structured so as to facilitate, and make more systematic, the task of the inspector. One suggestion has been to restrict to an absolute minimum the amount of information that the running process can access at any given moment. This is expressed by saying that the domain of protection in which the process runs is small. Unfortunately, if everything is done in software, the frequent change of domain which this approach makes necessary leads to unacceptable loss of performance. Attention was accordingly directed to providing hardware support for domain switching. The amount of information available to a process decreased as it moved from the inner to the outer rings. Unfortunately, the hierarchical model of protection which this implied is fundamentally flawed, and it was found that rings of protection were little improvement if any on the simple system of a privileged and an unprivileged mode. Unfortunately many key ideas such as exceptions handing, built-in strings, etc were dropped. Multics also was a pioneer in computer security, being essentially an opposite of Unix. In the early 60s, IBM was struggling to define its technical direction. The company had identified a problem with its past computer offerings: Each new product family, and each new generation of technology, forced customers to wrestle with an entirely new set of technical specifications. This was not, of course, unique to IBM. All computer vendors seemed to begin each new system with a "clean sheet" design. IBM saw this as both a problem and an opportunity. The cost of software migration was an increasing barrier to hardware sales. Customers could not afford to upgrade their computers, and IBM wanted to change this. IBM embarked on a very risky undertaking: Buying a bigger CPU also meant buying new printers, card readers, tape drives, etc. Customers would be able to "mix and match" to meet current needs; and they could confidently upgrade their systems in the future, without the need to rewrite all their software applications. IBM took on one of the largest and most ambitious engineering projects in history, and in the process discovered diseconomies of scale and the mythical man-month. Extensive literature on the period, such as that by Fred Brooks , illustrate the pitfalls. Moreover, time-sharing was new ground. Many of the concepts involved, such as virtual memory , remained unproven. Project MAC researchers were crushed and angered by this decision. IBM fully expected to win the Project MAC competition, and to retain its perceived lead in scientific computing and time-sharing. IBM had received intelligence that MIT was leaning toward the GE proposal, which was for a modified series computer with virtual memory hardware and other enhancements; this would eventually become the GE GE was prepared to make a large commitment to time-sharing, while IBM was seen as obstructive. As Corbato noted in his Turing lecture: Even the name was a joke. His strategy was clear. Start small and build up the ideas one by one as he saw how to implement them well. As we all know, UNIX has evolved and become immensely successful as the system of choice for workstations. And in this environment it quickly became the dominant programming language on mainframes in the USSR, far outpacing Cobol and Fortran that still dominated the mainframe arena in the USA and other Western countries. So here analogy with Perl hold perfectly. Conway and Thomas R. Here again the complexity and the costs of compilers were huge negative factor. Here is additional info from Wikipedia: It was heavily used by Daisy Systems for electronic design automation software on the "Logician" family of special-purpose workstations. It happened in Additional data types and attributes corresponding to common PC data types e. Improvements in readability of programs â€” often rendering implied usages explicit e. Along with natural complexity of the language there were couple of features that unnecessary complicated the creation of compiler: Which is of course unacceptable even in case of debugging compiler. Stings implementation was an example of premature optimization: Complex non-orthogonal programming language rarely became hugely popular. Popularity is reserved for simplistic, dumb-down languages. Cobol, Basic, Pascal and Java popularity are primary examples here. All of them are dull uninventive languages designed for novices with Pascal explicitly designed as for teaching programming at universities.

Chapter 7 : Software Security is a Programming Languages Issue - The PL Enthusiast

Programming Language/System (PL/S) is a language developed by IBM to replace assembly language. It is a machine language. Programming Language One (PL/I) is the base of this language.

Chapter 8 : Teaching Programming Languages - The PL Enthusiast

In CS we illustrate common vulnerability classes and how they can be defended against by the language (e.g., by using those languages, like Rust, that are type safe) and programming patterns (e.g., by validating untrusted input).

Chapter 9 : Programming in PL/I

Functional programming is the largest unit in , covering weeks of lectures and 3 programming projects (with two of them split in half). The students complete some warmup programming exercises, and later use OCaml to build a "Small C" language interpreter and a regular expression interpreter/compiler.