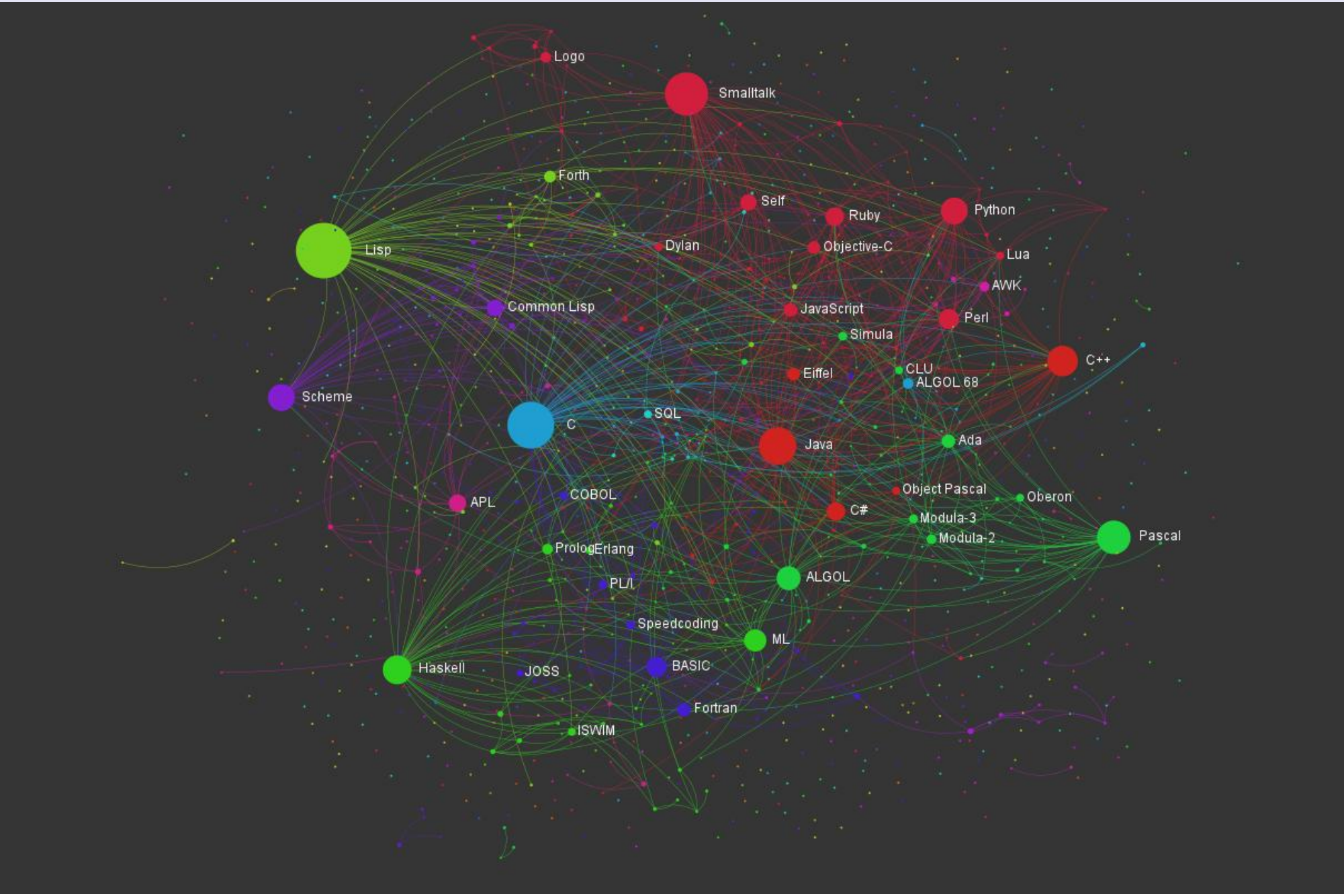
Logo
Smalltalk
Forth
Self
Ruby
Python
Lisp
Dylan
Objective-C
Lua
AWK
Common Lisp
JavaScript
Perl
Simula
C++
Eiffel
CLU
ALGOL 68
Scheme
C
SQL
Java
Ada
APL
COBOL
Object Pascal
Oberon
C#
Modula-3
Modula-2
Pascal
Prolog
Erlang
ALGOL
PL/I
Speedcoding
ML
Haskell
JOSS
BASIC
Fortran
ISWIM

From

`http://exploringdata.github.io/vis/programming-languages-influence-network/`

Introduction

In this part of the course we have:

- more on language families
- a detailed look at one family

More Language Families

We continue looking at general classes of languages, with less emphasis on specific examples of each

More Language Families

We continue looking at general classes of languages, with less emphasis on specific examples of each

You are expected to go and fill in the details yourself

More Language Families

We continue looking at general classes of languages, with less emphasis on specific examples of each

You are expected to go and fill in the details yourself

In each class there are usually dozens of members: writing new languages is easy these days

More Language Families

We continue looking at general classes of languages, with less emphasis on specific examples of each

You are expected to go and fill in the details yourself

In each class there are usually dozens of members: writing new languages is easy these days

Designing *good* and *useful* languages is much harder

More Language Families

We continue looking at general classes of languages, with less emphasis on specific examples of each

You are expected to go and fill in the details yourself

In each class there are usually dozens of members: writing new languages is easy these days

Designing *good* and *useful* languages is much harder

And usually unnecessary

Language Families

You have seen

- C: procedural
- Lisp: functional
- Python: scripting
- Java: procedural and object oriented
- etc.

Language Families

You have seen

- C: procedural
- Lisp: functional
- Python: scripting
- Java: procedural and object oriented
- etc.

Next semester you will see logic and declarative languages (Prolog, ASP)

Language Families

Feet

- C: you shoot yourself in the foot

Language Families

Feet

- C: you shoot yourself in the foot
- Lisp: You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds...

Language Families

Feet

- C: you shoot yourself in the foot
- Lisp: You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds. . .
- Prolog: You tell your program you want to be shot in the foot. The program figures out how to do it, but the syntax doesn't allow it to explain

Language Families

Feet

- C: you shoot yourself in the foot
- Lisp: You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds. . .
- Prolog: You tell your program you want to be shot in the foot. The program figures out how to do it, but the syntax doesn't allow it to explain
- Python: You try to shoot yourself in the foot but you just keep hitting the whitespace between your toes

Language Families

Feet

- Java: You locate the Gun class, but discover that the Bullet class is abstract, so you extend it and write the missing part of the implementation. Then you implement the ShootAble interface for your foot, and recompile the Foot class. The interface lets the bullet call the doDamage method on the Foot, so the Foot can damage itself in the most effective way. Now you run the program, and call the doShoot method on the instance of the Gun class. First the Gun creates an instance of Bullet, which calls the doFire method on the Gun. The Gun calls the hit(Bullet) method on the Foot, and the instance of Bullet is passed to the Foot. But this causes an IllegalHitByBullet exception to be thrown, and you die

Language Families

Feet

- Cobol: USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied

Language Families

Feet

- Cobol: USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied
- Fortran: You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-handling facility

Language Families

Feet

- Cobol: USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied
- Fortran: You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-handling facility
- APL: You hear a gunshot and there's a hole in your foot, but you don't remember enough linear algebra to understand what happened.

Language Families

Feet

- Cobol: USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied
- Fortran: You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-handling facility
- APL: You hear a gunshot and there's a hole in your foot, but you don't remember enough linear algebra to understand what happened.
- Snobol: If you succeed, shoot yourself in the left foot. If you fail, shoot yourself in the right foot

Language Families

Feet

Continuing exercise: go and read further around these (and other) languages to discover why they have these descriptions

Language Families

Feet

Continuing exercise: go and read further around these (and other) languages to discover why they have these descriptions

Exercise for advanced students: make up jokes for the missing ones and funnier versions for existing ones

Language Families

There are hundreds of languages out there

Language Families

There are hundreds of languages out there

How do we choose which to use?

Language Families

There are hundreds of languages out there

How do we choose which to use?

Sometimes we are told by the boss, or have to fit with an existing project

Language Families

There are hundreds of languages out there

How do we choose which to use?

Sometimes we are told by the boss, or have to fit with an existing project

Sometimes we only have a restricted choice

Language Families

If we have any choice we need to have some criteria to guide the choice

Language Families

If we have any choice we need to have some criteria to guide the choice

We need to able to

- identify and assess characteristics of a given language

Language Families

If we have any choice we need to have some criteria to guide the choice

We need to able to

- identify and assess characteristics of a given language
- recognise similarities between languages

Language Families

If we have any choice we need to have some criteria to guide the choice

We need to able to

- identify and assess characteristics of a given language
- recognise similarities between languages
- recognise if a feature is unique to a language

Language Families

If we have any choice we need to have some criteria to guide the choice

We need to able to

- identify and assess characteristics of a given language
- recognise similarities between languages
- recognise if a feature is unique to a language
- take concepts from one language to another (learn one, learn 'em all)

Language Families

“If you can’t say it you can’t think it” (Orwell/Wittgenstein)

Language Families

“If you can’t say it you can’t think it” (Orwell/Wittgenstein)

Having more concepts allows more flexibility: if there is no array construct in the language, you are restricted in what you can do easily

Language Families

“If you can’t say it you can’t think it” (Orwell/Wittgenstein)

Having more concepts allows more flexibility: if there is no array construct in the language, you are restricted in what you can do easily

“Those who cannot remember the past are condemned to repeat it” (George Santayana)

Language Families

“If you can’t say it you can’t think it” (Orwell/Wittgenstein)

Having more concepts allows more flexibility: if there is no array construct in the language, you are restricted in what you can do easily

“Those who cannot remember the past are condemned to repeat it” (George Santayana)

Avoid re-implementation and old mistakes: wise people learn from the mistakes of others

Language Families

So to do this we classify language into families

Language Families

So to do this we classify language into families

Members of a family have some things in common

Language Families

So to do this we classify language into families

Members of a family have some things in common

This is possible because language designers rarely have new ideas, they just borrow from other languages

Language Families

So to do this we classify language into families

Members of a family have some things in common

This is possible because language designers rarely have new ideas, they just borrow from other languages

Families are **not exclusive**, a language can sit comfortably in more than one family

Language Families

We shall be going through some popular families and for each we will look at:

Language Families

We shall be going through some popular families and for each we will look at:

Purpose: what these languages are *generally* used for

Language Families

We shall be going through some popular families and for each we will look at:

Purpose: what these languages are *generally* used for

Of course, you can do pretty much anything computable in any language, but certain languages make certain things easier

Language Families

We shall be going through some popular families and for each we will look at:

Purpose: what these languages are *generally* used for

Of course, you can do pretty much anything computable in any language, but certain languages make certain things easier

Or harder, if you are trying to avoid errors

Language Families

Examples: some languages that are generally regarded as being in this family

Language Families

Examples: some languages that are generally regarded as being in this family

Again, many languages can live in more than one family

Language Families

Examples: some languages that are generally regarded as being in this family

Again, many languages can live in more than one family

Some people would be upset if we called Java procedural (it is), but its main distinguishing feature is being object oriented

Languages Families

Notable features: general comments

Languages Families

Notable features: general comments

Many languages were designed for a purpose; some were not really designed

Languages Families

Notable features: general comments

Many languages were designed for a purpose; some were not really designed

What designers think as important has changed over the years, as knowledge of CS has increased and computers have developed

Languages Families

Notable features: general comments

Many languages were designed for a purpose; some were not really designed

What designers think as important has changed over the years, as knowledge of CS has increased and computers have developed

Older languages tend to have different domains of competence than newer languages

Languages Families

Notable features: general comments

Many languages were designed for a purpose; some were not really designed

What designers think as important has changed over the years, as knowledge of CS has increased and computers have developed

Older languages tend to have different domains of competence than newer languages

Often the aim of a language is *control of complexity*: how can I write a bigger program that is still *correct*?

Language Families

Sometimes the aim is to solve a particular problem or class of problems

Language Families

Sometimes the aim is to solve a particular problem or class of problems

For example: symbolic algebra; logic; business; string manipulation; drawing pictures; manipulating Web pages

Language Families

Sometimes the aim is to solve a particular problem or class of problems

For example: symbolic algebra; logic; business; string manipulation; drawing pictures; manipulating Web pages

The number of *general purpose* languages is relatively small

Language Families

Sometimes the aim is to solve a particular problem or class of problems

For example: symbolic algebra; logic; business; string manipulation; drawing pictures; manipulating Web pages

The number of *general purpose* languages is relatively small

We start by looking at the earlier, unstructured, languages

Unstructured Languages

Purpose: general programming

Examples: assembler, early Basic, ...

Notable features: lack of language features to help structure large programs

Unstructured Languages

Feet

- Assembly: You try to shoot yourself in the foot only to discover you must first reinvent the gun, the bullet, and your foot. After that's done, you pull the trigger, the gun beeps several times, then crashes.

Unstructured Languages

Feet

- Assembly: You try to shoot yourself in the foot only to discover you must first reinvent the gun, the bullet, and your foot. After that's done, you pull the trigger, the gun beeps several times, then crashes.
- Basic: Shoot yourself in the foot with a water pistol. On big systems, continue until entire lower body is waterlogged

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

The programs written in such languages tended to be small, as computers were small, and so they were manageable

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

The programs written in such languages tended to be small, as computers were small, and so they were manageable

Up to a point

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

The programs written in such languages tended to be small, as computers were small, and so they were manageable

Up to a point

It was soon discovered that you can't write bigger programs in this way

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

The programs written in such languages tended to be small, as computers were small, and so they were manageable

Up to a point

It was soon discovered that you can't write bigger programs in this way

A language needed *structure* to help the programmer

Unstructured Languages

These languages (it is arguable whether assembly is even a language) were used before there were any clear ideas in CS on what was needed to write a large, correct program

The programs written in such languages tended to be small, as computers were small, and so they were manageable

Up to a point

It was soon discovered that you can't write bigger programs in this way

A language needed *structure* to help the programmer

To some extent, the history of computing languages is the history of the varied attempts to provide that structure

Procedural Languages

Purpose: general programming

Examples: C, Fortran, Cobol, Pascal, Algol, later Basic, Maple, . . .

Notable features: use of functions (procedures) to provide structure and control complexity

Procedural Languages

Feet

- Pascal: The compiler won't let you shoot yourself in the foot

Procedural Languages

Feet

- Pascal: The compiler won't let you shoot yourself in the foot
- Algol: You shoot yourself in the foot with a musket. The musket is aesthetically fascinating and the wound baffles the adolescent medic in the emergency room

Procedural Languages

Feet

- Pascal: The compiler won't let you shoot yourself in the foot
- Algol: You shoot yourself in the foot with a musket. The musket is aesthetically fascinating and the wound baffles the adolescent medic in the emergency room
- Algol 68: You mildly deprocedure the gun, the bullet gets firmly dereferenced, and your foot is strongly coerced to void

Procedural Languages

Feet

- Pascal: The compiler won't let you shoot yourself in the foot
- Algol: You shoot yourself in the foot with a musket. The musket is aesthetically fascinating and the wound baffles the adolescent medic in the emergency room
- Algol 68: You mildly deprocedure the gun, the bullet gets firmly dereferenced, and your foot is strongly coerced to void
- Maple: A ShootFoot function was not implemented in Release n, but will be included in Release n+1. Meanwhile, you may purchase Release n at the Release n+1 price

Procedural Languages

Procedures, subroutines and functions were soon invented as they encapsulate an idea in a localised chunk of code

Procedural Languages

Procedures, subroutines and functions were soon invented as they encapsulate an idea in a localised chunk of code

If done correctly

Procedural Languages

Procedures, subroutines and functions were soon invented as they encapsulate an idea in a localised chunk of code

If done correctly

Procedural languages came very early (Fortran) and are still used widely today (C)

Procedural Languages

Procedures, subroutines and functions were soon invented as they encapsulate an idea in a localised chunk of code

If done correctly

Procedural languages came very early (Fortran) and are still used widely today (C)

They are very successful and many large systems have been written using them

Logic Languages

Purpose: Logic programming

Examples: Prolog, ASP, ...

Notable features: don't describe *how* to do something, just what you want as an answer

Logic Languages

Logic languages have also been around a long time, but are much less popular than other families

Logic Languages

Logic languages have also been around a long time, but are much less popular than other families

Many people have difficulty using them for general purpose programming

Logic Languages

Logic languages have also been around a long time, but are much less popular than other families

Many people have difficulty using them for general purpose programming

For logic problems, of course, they are excellent

Functional Languages

Purpose: general programming, symbolic programming

Examples: Lisp, Haskell, ML, Erlang, Scala, ...

Notable features: use of higher order functions to provide structure and control complexity

Functional Languages

Feet

- Haskell: You spend several hours creating a new copy of the Universe which is identical to the existing one except your foot has a hole in it. You then hear that it can be done more elegantly with Dyadic Functile Hyper-Arrows, but the very act of reading some of the included sample code causes one of your metatarsals to explode

Functional Languages

Feet

- Haskell: You spend several hours creating a new copy of the Universe which is identical to the existing one except your foot has a hole in it. You then hear that it can be done more elegantly with Dyadic Functile Hyper-Arrows, but the very act of reading some of the included sample code causes one of your metatarsals to explode
- Haskell (2): You appear to have successfully shot yourself in the foot, but you feel no pain. Until you look at your foot

Functional Languages

Feet

- Haskell: You spend several hours creating a new copy of the Universe which is identical to the existing one except your foot has a hole in it. You then hear that it can be done more elegantly with Dyadic Functile Hyper-Arrows, but the very act of reading some of the included sample code causes one of your metatarsals to explode
- Haskell (2): You appear to have successfully shot yourself in the foot, but you feel no pain. Until you look at your foot
- Erlang: whenever you shoot your foot off, you just grow more feet

Functional Languages

Feet

- Haskell: You spend several hours creating a new copy of the Universe which is identical to the existing one except your foot has a hole in it. You then hear that it can be done more elegantly with Dyadic Functile Hyper-Arrows, but the very act of reading some of the included sample code causes one of your metatarsals to explode
- Haskell (2): You appear to have successfully shot yourself in the foot, but you feel no pain. Until you look at your foot
- Erlang: whenever you shoot your foot off, you just grow more feet
- Scala: You can't find anyone who knows how to shoot you in the foot

Functional Languages

Feet

- ML: You program a structure for your foot, the gun, and the bullet, complete with associated signatures and function definitions. After two hours of laborious typing, forgetting of semicolons, and searching old Comp Sci textbooks for the definition of such phrases as “polymorphic dynamic objective typing system”, as well as an additional hour for brushing up on the lambda calculus, you run the program and the interpreter tells you that the pattern-match between your foot and the bullet is nonexhaustive. You feel a slight tingling pain, but no bullethole appears in your foot because your program did not allow for side-effecting statements

Functional Languages

While functional languages themselves have yet to gain widespread use, the ideas they have generated are used daily

Functional Languages

While functional languages themselves have yet to gain widespread use, the ideas they have generated are used daily

As parallel computers become more popular there may well be a re-examination of functional style programming as it naturally supports (implicit) parallelism

Functional Languages

While functional languages themselves have yet to gain widespread use, the ideas they have generated are used daily

As parallel computers become more popular there may well be a re-examination of functional style programming as it naturally supports (implicit) parallelism

And mainstream languages like Java and C++ are adopting functional concepts like lambdas and iterators