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

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

Some languages are designed to make good programming easier, while some are designed to make bad programming harder

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 has procedural features), 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 new language is *control of complexity*: how can I write a bigger program that is still *correct*?

Language Families

Many languages were designed to solve a particular problem or class of problems, e.g.: symbolic algebra; logic; business; string manipulation; drawing pictures; manipulating Web pages

Language Families

Many languages were designed to solve a particular problem or class of problems, e.g.: symbolic algebra; logic; business; string manipulation; drawing pictures; manipulating Web pages

The number of *general purpose* languages is relatively small

Language Families

Many languages were designed to solve a particular problem or class of problems, e.g.: 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: assembly language, early Basic, . . .

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

Unstructured Languages

Feet

- Assembly language: 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 language: 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 language 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 language 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 language 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

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

Unstructured Languages

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

Languages needed *structuring mechanisms* to help the programmer

Unstructured Languages

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

Languages needed *structuring mechanisms* to help the programmer

To some extent, the history of computing languages is the history of the varied attempts to provide those mechanisms

Procedural Languages

Purpose: general programming

Examples: C, Fortran, Cobol, Pascal, Oberon, Algol, Ada, later Basic, . . .

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

Procedural Languages

Feet

- 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

- 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

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

Procedural Languages

Feet

- 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
- Pascal: The compiler won't let you shoot yourself in the foot
- Oberon: The gun keeps jamming and the bullets are probably blanks, so you kick the computer and break your foot

Procedural Languages

Feet

- Ada: If you are dumb enough to actually use this language, the United States Department of Defense will kidnap you, stand you up in front of a firing squad, and tell the soldiers, “Shoot at the feet.”

Procedural Languages

Feet

- Ada: If you are dumb enough to actually use this language, the United States Department of Defense will kidnap you, stand you up in front of a firing squad, and tell the soldiers, “Shoot at the feet.”
- Ada (2): After correctly packing your foot, you attempt to concurrently load the gun, pull the trigger, scream, and confidently aim at your foot knowing it is safe. However the cordite in the round does an Unchecked Conversion, fires and shoots you in the foot anyway.

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, Lisp) and are still used widely today (C, Lisp and Fortran)

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, Lisp) and are still used widely today (C, Lisp and Fortran)

They are very successful and many large systems (millions of lines of code) have been written using them, particularly 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, Lisp) and are still used widely today (C, Lisp and Fortran)

They are very successful and many large systems (millions of lines of code) have been written using them, particularly C

> *[C is] like juggling chainsaws*

Linus Torvalds, overseer of the Linux kernel

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

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

Logic Languages

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

```
man(X) :- mortal(X)
man(socrates)
?- mortal(socrates)
```

Logic Languages

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

```
man(X) :- mortal(X)
man(socrates)
?- mortal(socrates)
-> Yes
```

Logic Languages

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

```
man(X) :- mortal(X)
man(socrates)
?- mortal(socrates)
-> Yes
```

“Is anything mortal?”

Logic Languages

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

```
man(X) :- mortal(X)
man(socrates)
?- mortal(socrates)
-> Yes
```

“Is anything mortal?”

```
?- mortal(X)
```

Logic Languages

“All men are mortal”. “Socrates is a man”. Is Socrates mortal?

```
man(X) :- mortal(X)
man(socrates)
?- mortal(socrates)
-> Yes
```

“Is anything mortal?”

```
?- mortal(X)
-> X = socrates
```

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

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

Much used in early AI, before deep learning became the only accepted way of solving every problem

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

Much used in early AI, before deep learning became the only accepted way of solving every problem

Exercise Read about how Prolog was used (in 2023) to find a set of 27 lottery tickets that is guaranteed a win in the UK lottery (N.B. *not* necessarily a profit!)

Functional Languages

Purpose: general programming, symbolic programming

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

Notable features: use of higher order functions to provide structure and control complexity; avoidance of side-effects; avoidance of variable update and value modification

Functional Languages

Purpose: general programming, symbolic programming

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

Notable features: use of higher order functions to provide structure and control complexity; avoidance of side-effects; avoidance of variable update and value modification

Functions are first class: they are values in their own right and can be passed in other functions as arguments and returned from functions as values, sometimes even constructed at runtime

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Variously called *pointfree* programming, *tacit* programming, or *combinator* programming)

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Variously called *pointfree* programming, *tacit* programming, or *combinator* programming)

Suppose you want to add 1 to each element of a vector

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Variously called *pointfree* programming, *tacit* programming, or *combinator* programming)

Suppose you want to add 1 to each element of a vector

In many languages you would use a `for` loop: “take the first value, add 1; take the second value, add 1; etc.”

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Variously called *pointfree* programming, *tacit* programming, or *combinator* programming)

Suppose you want to add 1 to each element of a vector

In many languages you would use a `for` loop: “take the first value, add 1; take the second value, add 1; etc.”

The functional style code is essentially: “add 1 to all values in this vector”

Functional Languages

Datastructures are treated holistically, rather than element-by-element

Variously called *pointfree* programming, *tacit* programming, or *combinator* programming)

Suppose you want to add 1 to each element of a vector

In many languages you would use a `for` loop: “take the first value, add 1; take the second value, add 1; etc.”

The functional style code is essentially: “add 1 to all values in this vector”

Some functional style languages don’t even have loops

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

Feet

- Scheme: 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 the gun with which you shoot yourself in the appendage which holds. . . but none of the other appendages are aware of this happening.

Functional Languages

Feet

- Scheme: 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 the gun with which you shoot yourself in the appendage which holds. . . but none of the other appendages are aware of this happening.
- Scheme (2): Scheme does not provide a gun as it can be constructed from more fundamental concepts. Nor 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

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

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 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 parallelism

And mainstream languages like Java and C++ are adopting functional concepts like closures, maps and iterators

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 parallelism

And mainstream languages like Java and C++ are adopting functional concepts like closures, maps and iterators

These concepts are higher-level and more “natural” (closer to the way we normally think)

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 parallelism

And mainstream languages like Java and C++ are adopting functional concepts like closures, maps and iterators

These concepts are higher-level and more “natural” (closer to the way we normally think)

Or would be if we hadn’t been taught the less natural OO and procedural styles

Macro languages

Purpose: to improve readability of other code, abstraction, textual manipulation

Examples: Cpp, LaTeX, M4, macros in Lisp

Notable features: usually lexical (character or text) based, with some exceptions that are syntax based (Lisp, Rust)

Macro languages

Purpose: to improve readability of other code, abstraction, textual manipulation

Examples: Cpp, LaTeX, M4, macros in Lisp

Notable features: usually lexical (character or text) based, with some exceptions that are syntax based (Lisp, Rust)

These languages usually say “if you see something like this, replace it with that”; used in particular in program source code, with the output then processed by the compiler

Macro languages

Purpose: to improve readability of other code, abstraction, textual manipulation

Examples: Cpp, LaTeX, M4, macros in Lisp

Notable features: usually lexical (character or text) based, with some exceptions that are syntax based (Lisp, Rust)

These languages usually say “if you see something like this, replace it with that”; used in particular in program source code, with the output then processed by the compiler

So, usually, code to be executed in the *compilation phase*, before the main compiler, rather than at runtime

Macro languages

Feet

- LaTeX:

```
\documentclass[12pt]{article}
\usepackage{latexgun,latexshoot}
\begin{document}
See how easy it is to shoot yourself in the foot? \\
\gun[leftfoot]{shoot} \\
\pain
\end{document}
```

Macro languages

These are used widely, in a huge variety of contexts

Macro languages

These are used widely, in a huge variety of contexts

Commonly used in conjunction with another language, e.g., the C preprocessor (Cpp) modifying code before passing it to the C compiler

Macro languages

These are used widely, in a huge variety of contexts

Commonly used in conjunction with another language, e.g., the C preprocessor (Cpp) modifying code before passing it to the C compiler

C code $\xrightarrow[\text{preprocessor}]{}$ C code $\xrightarrow[\text{compiler}]{}$ machine code

Macro languages

These are used widely, in a huge variety of contexts

Commonly used in conjunction with another language, e.g., the C preprocessor (Cpp) modifying code before passing it to the C compiler

$$\text{C code} \xrightarrow[\text{preprocessor}]{} \text{C code} \xrightarrow[\text{compiler}]{} \text{machine code}$$

Not often thought about in great detail, but used to great effect

Macro languages

These are used widely, in a huge variety of contexts

Commonly used in conjunction with another language, e.g., the C preprocessor (Cpp) modifying code before passing it to the C compiler

$$\text{C code} \xrightarrow[\text{preprocessor}]{} \text{C code} \xrightarrow[\text{compiler}]{} \text{machine code}$$

Not often thought about in great detail, but used to great effect

Particularly *conditional macros* whose expansion depends on other factors

Macro languages

These are used widely, in a huge variety of contexts

Commonly used in conjunction with another language, e.g., the C preprocessor (Cpp) modifying code before passing it to the C compiler

$$\text{C code} \xrightarrow[\text{preprocessor}]{} \text{C code} \xrightarrow[\text{compiler}]{} \text{machine code}$$

Not often thought about in great detail, but used to great effect

Particularly *conditional macros* whose expansion depends on other factors

Or when the source needs some help writing, e.g., a large number of similar bits of code

Macro languages

So, in C, we can write macro code like

```
#ifdef SMALLINT
#define NUMBER short
#else
#define NUMBER int
#endif
```

Macro languages

So, in C, we can write macro code like

```
#ifdef SMALLINT
#define NUMBER short
#else
#define NUMBER int
#endif
```

In the C code following this the text token "`NUMBER`" is replaced by the text "`short`" or "`int`" as appropriate

Macro languages

So, in C, we can write macro code like

```
#ifdef SMALLINT
#define NUMBER short
#else
#define NUMBER int
#endif
```

In the C code following this the text token "`NUMBER`" is replaced by the text "`short`" or "`int`" as appropriate

Then if we use `NUMBER` everywhere in our code
`NUMBER x; ...`
it takes only a single change to make our code use `short` rather than `int`: very useful for source code portability between architectures

C

```
#define _ F-->00||-F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_
            _-_-_-_
}
```

An enthusiastic use of C macros by Brian Westley

Macro languages

The C preprocessor language is quite different from the C language, and works on purely a textual level: it doesn't "understand" the structure of C at all

Macro languages

The C preprocessor language is quite different from the C language, and works on purely a textual level: it doesn't "understand" the structure of C at all

And can actually be used to preprocess any text file, not just C

Macro languages

The C preprocessor language is quite different from the C language, and works on purely a textual level: it doesn't "understand" the structure of C at all

And can actually be used to preprocess any text file, not just C

In contrast to most languages, Lisp macros are not text based, but expression based: expressions are replaced by expressions

Macro languages

The C preprocessor language is quite different from the C language, and works on purely a textual level: it doesn't "understand" the structure of C at all

And can actually be used to preprocess any text file, not just C

In contrast to most languages, Lisp macros are not text based, but expression based: expressions are replaced by expressions

And the macro language is Lisp itself, not a separate language: in Lisp, data and program have identical representations, so programs are data and data can be a program!

Macro languages

The C preprocessor language is quite different from the C language, and works on purely a textual level: it doesn't "understand" the structure of C at all

And can actually be used to preprocess any text file, not just C

In contrast to most languages, Lisp macros are not text based, but expression based: expressions are replaced by expressions

And the macro language is Lisp itself, not a separate language: in Lisp, data and program have identical representations, so programs are data and data can be a program!

The full power of Lisp applies to macroexpansion: code that manipulates code

Macro languages

LaTeX

Sometimes macros are used in their own right, e.g., LaTeX, the typesetting language is macro based

Macro languages

LaTeX

Sometimes macros are used in their own right, e.g., LaTeX, the typesetting language is macro based

These slides are written in LaTeX

Macro languages

LaTeX

Sometimes macros are used in their own right, e.g., LaTeX, the typesetting language is macro based

These slides are written in LaTeX

```
\begin{frame}
\frametitle{Macro languages}
\framesubtitle{\LaTeX}

Sometimes macros are used in their own right, e.g.,
\LaTeX, the typesetting language is macro based

\pc{These slides are written in \LaTeX}
\end{frame}
```

Macro languages

LaTeX

Here the basic datatype is text and you define macros as convenient shorthands for things you like to do to that text

Macro languages

LaTeX

Here the basic datatype is text and you define macros as convenient shorthands for things you like to do to that text

For example, a new chapter needs a new page, a big heading, lots of space before the next text

Macro languages

LaTeX

Here the basic datatype is text and you define macros as convenient shorthands for things you like to do to that text

For example, a new chapter needs a new page, a big heading, lots of space before the next text

So you define a macro, say `\chapter`, that does all this (or you use a library definition)

Macro languages

LaTeX

Here the basic datatype is text and you define macros as convenient shorthands for things you like to do to that text

For example, a new chapter needs a new page, a big heading, lots of space before the next text

So you define a macro, say `\chapter`, that does all this (or you use a library definition)

And if you want to change the layout, you just change the `\chapter` macro

Macro languages

LaTeX

Here the basic datatype is text and you define macros as convenient shorthands for things you like to do to that text

For example, a new chapter needs a new page, a big heading, lots of space before the next text

So you define a macro, say `\chapter`, that does all this (or you use a library definition)

And if you want to change the layout, you just change the `\chapter` macro

Exercise Compare with WYSIWYG word processors

Scripting Languages

Purpose: control of other elements of a system, e.g., programs, “glue” to join elements together

Examples: DOS batch, sh, Python, Sed, Perl, Ruby, JavaScript . . .

Notable features: not particularly good at classical number crunching; generally lots of string processing and process manipulation

Scripting Languages

Purpose: control of other elements of a system, e.g., programs, “glue” to join elements together

Examples: DOS batch, sh, Python, Sed, Perl, Ruby, JavaScript . . .

Notable features: not particularly good at classical number crunching; generally lots of string processing and process manipulation

They are called “scripts” as they are (were originally) lists of things to be done

Scripting Languages

Purpose: control of other elements of a system, e.g., programs, “glue” to join elements together

Examples: DOS batch, sh, Python, Sed, Perl, Ruby, JavaScript . . .

Notable features: not particularly good at classical number crunching; generally lots of string processing and process manipulation

They are called “scripts” as they are (were originally) lists of things to be done

Often, but not exclusively, interpreted rather than compiled

Scripting Languages

Feet

- DOS batch: You aim the gun at your foot and pull the trigger, but only a weak gust of warm air hits your foot

Scripting Languages

Feet

- DOS batch: You aim the gun at your foot and pull the trigger, but only a weak gust of warm air hits your foot
- Sh, csh, bash: You can't remember the syntax for anything so you spend five hours reading man pages then your foot falls asleep. You then shoot the computer and switch to Perl

Scripting Languages

Feet

- DOS batch: You aim the gun at your foot and pull the trigger, but only a weak gust of warm air hits your foot
- Sh, csh, bash: You can't remember the syntax for anything so you spend five hours reading man pages then your foot falls asleep. You then shoot the computer and switch to Perl
- Perl: You shoot yourself in the foot, but nobody can understand how you did it. Six months later, neither can you

Scripting Languages

Feet

- DOS batch: You aim the gun at your foot and pull the trigger, but only a weak gust of warm air hits your foot
- Sh, csh, bash: You can't remember the syntax for anything so you spend five hours reading man pages then your foot falls asleep. You then shoot the computer and switch to Perl
- Perl: You shoot yourself in the foot, but nobody can understand how you did it. Six months later, neither can you
- Ruby: Your foot is ready to be shot in roughly five minutes, but you just can't find anywhere to shoot it

Scripting Languages

Feet

- DOS batch: You aim the gun at your foot and pull the trigger, but only a weak gust of warm air hits your foot
- Sh, csh, bash: You can't remember the syntax for anything so you spend five hours reading man pages then your foot falls asleep. You then shoot the computer and switch to Perl
- Perl: You shoot yourself in the foot, but nobody can understand how you did it. Six months later, neither can you
- Ruby: Your foot is ready to be shot in roughly five minutes, but you just can't find anywhere to shoot it
- JavaScript: You've perfected a robust, rich user experience for shooting yourself in the foot. You then find that bullets are disabled on your gun

Scripting Languages

Also a widely used family

Scripting Languages

Also a widely used family

The original scripts were the job control languages for the early mainframes

Scripting Languages

Also a widely used family

The original scripts were the job control languages for the early mainframes

For example, IBM's JCL was used to describe a list of programs to be loaded and run: recall batch processing

Scripting Languages

Also a widely used family

The original scripts were the job control languages for the early mainframes

For example, IBM's JCL was used to describe a list of programs to be loaded and run: recall batch processing

To compile and run them, you (the computer) just follow the script

Scripting Languages

Feet

- JCL: You send your foot down to MIS with a 4000-page document explaining how you want it to be shot. Three years later, your foot comes back deep-fried

Scripting Languages

Feet

Job control languages are not a thing of the past: modern large supercomputers are typically still managed using job scripts

Scripting Languages

Feet

Job control languages are not a thing of the past: modern large supercomputers are typically still managed using job scripts

Exercise Read about PBS and SLURM

Scripting Languages

Sh

The Unix command line language, originally `sh` (the Bourne Shell) and variants such as `csh` (C shell), lately `bash` (the Bourne Again shell)

Scripting Languages

Sh

The Unix command line language, originally `sh` (the Bourne Shell) and variants such as `csh` (C shell), lately `bash` (the Bourne Again shell)

Simple textual lists of things (programs) to be done

Scripting Languages

Sh

The Unix command line language, originally `sh` (the Bourne Shell) and variants such as `csh` (C shell), lately `bash` (the Bourne Again shell)

Simple textual lists of things (programs) to be done

```
#!/bin/sh
setxkbmap -option "compose:menu" -option "ctrl:nocaps"
dispwin -L
[ "$XAUTHORITY" ] && cp -f "$XAUTHORITY" ~/.Xauthority
```

Scripting Languages

Sh

Shell scripts are widely used to automate repetitive and complex tasks

Scripting Languages

Sh

Shell scripts are widely used to automate repetitive and complex tasks

Originally reasonably simple as languages but they have grown more complex in their abilities over time

Scripting Languages

Sh

Shell scripts are widely used to automate repetitive and complex tasks

Originally reasonably simple as languages but they have grown more complex in their abilities over time

Crucially, they do not require a GUI so can be deployed automatically over large numbers of machines

Scripting Languages

Sh

Shell scripts are widely used to automate repetitive and complex tasks

Originally reasonably simple as languages but they have grown more complex in their abilities over time

Crucially, they do not require a GUI so can be deployed automatically over large numbers of machines

Somewhat low-level, so not so good for more complex tasks, or less complex programmers