Functional Style

Does not updating state mean no updating variables?

Functional Style

Does not updating state mean no updating variables?

Correct!

Functional Style

Does not updating state mean no updating variables?

Correct!

What about things like

```
for (i = 0; i < 10; i++) ...
```

Functional Style

Does not updating state mean no updating variables?

Correct!

What about things like

```
for (i = 0; i < 10; i++) ...
```

`i` is of essence varying, updated every time around the loop

Functional Style

Does not updating state mean no updating variables?

Correct!

What about things like

```
for (i = 0; i < 10; i++) ...
```

`i` is of essence varying, updated every time around the loop

So we can't do loops like this

Functional Style

Does not updating state mean no updating variables?

Correct!

What about things like

```
for (i = 0; i < 10; i++) ...
```

`i` is of essence varying, updated every time around the loop

So we can't do loops like this

But we have to have repetition to have useful programs

Functional Style

Recursion

So we use *recursion*

Functional Style

Recursion

So we use *recursion*

Functional style programs use recursion as a fundamental tool

Functional Style

Recursion

So we use *recursion*

Functional style programs use recursion as a fundamental tool

This has consequences, as we shall see

Functional Style

Recursion

So we use *recursion*

Functional style programs use recursion as a fundamental tool

This has consequences, as we shall see

One is that it leads naturally to having functions as *first class objects*

Functional Style

Recursion

A *first class object* is an object (not in the OO sense, just some thing) that is treated equally with all others

Functional Style

Recursion

A *first class object* is an object (not in the OO sense, just some thing) that is treated equally with all others

In particular it can be

Functional Style

Recursion

A *first class object* is an object (not in the OO sense, just some thing) that is treated equally with all others

In particular it can be

- created and destroyed at runtime

Functional Style

Recursion

A *first class object* is an object (not in the OO sense, just some thing) that is treated equally with all others

In particular it can be

- created and destroyed at runtime
- passed into a function as an argument

Functional Style

Recursion

A *first class object* is an object (not in the OO sense, just some thing) that is treated equally with all others

In particular it can be

- created and destroyed at runtime
- passed into a function as an argument
- returned from a function as a result

Functional Style

Recursion

So we are going to need functions that manipulate functions: these are called *higher order* functions

Functional Style

Recursion

So we are going to need functions that manipulate functions: these are called *higher order* functions

We typically can have expressions involving functions

Functional Style

Recursion

So we are going to need functions that manipulate functions: these are called *higher order* functions

We typically can have expressions involving functions

```
(if x > 0.0 then sin else cos)(3.0)
```

Functional Style

Recursion

So we are going to need functions that manipulate functions: these are called *higher order* functions

We typically can have expressions involving functions

```
(if x > 0.0 then sin else cos)(3.0)
```

Other languages might do

```
if x > 0.0 then sin(3.0) else cos(3.0)
```

Functional Style

Recursion

So we are going to need functions that manipulate functions: these are called *higher order* functions

We typically can have expressions involving functions

```
(if x > 0.0 then sin else cos)(3.0)
```

Other languages might do

```
if x > 0.0 then sin(3.0) else cos(3.0)
```

Again, a trivial (and contrived) example of a much bigger idea

Functional Style

Recursion

Exercise. Investigate

```
double foo(double x) {
  return (x > 0.0 ? sin : cos)(3.0);
}
```

in C

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

But try iterating over a binary tree with a for loop. . .

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

But try iterating over a binary tree with a for loop. . .

With recursion it’s trivial

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

But try iterating over a binary tree with a for loop. . .

With recursion it’s trivial

List: do the current value; recurse on the rest of the list

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

But try iterating over a binary tree with a for loop. . .

With recursion it’s trivial

List: do the current value; recurse on the rest of the list

Tree: do the current value; recurse on the left subtree; recurse on the right subtree

Functional Style

Recursion

Anything that can be done with iteration (“for loops”) can be done with recursion

But try iterating over a binary tree with a for loop. . .

With recursion it’s trivial

List: do the current value; recurse on the rest of the list

Tree: do the current value; recurse on the left subtree; recurse on the right subtree

We will find that the idea of separating the action of traversal of a datastructure from the operation on the elements of the datastructure is prominent in the functional style

Functional Style

There are many functional style languages out there

Functional Style

There are many functional style languages out there

Some encourage the functional style, but let you do OO or procedural style, too

Functional Style

There are many functional style languages out there

Some encourage the functional style, but let you do OO or procedural style, too

Others enforce the functional style, e.g., by not having assignment (variable update)

Functional Style

There are many functional style languages out there

Some encourage the functional style, but let you do OO or procedural style, too

Others enforce the functional style, e.g., by not having assignment (variable update)

Some started off as procedural and moved towards functional

Functional Style

There are many functional style languages out there

Some encourage the functional style, but let you do OO or procedural style, too

Others enforce the functional style, e.g., by not having assignment (variable update)

Some started off as procedural and moved towards functional

Some were designed from scratch as functional

Functional Style

Just a few names

- Lisp/Scheme (1959)
- APL (1964) “A Programming Language”
- ISWIM (1966) “If you See What I Mean”
- SASL (1972) “St. Andrews Static Language”
- ML/SML (1973) “Meta Language”
- Hope (1980)
- KRC (1981) “Kent Recursive Calculator”
- Miranda (1985)
- Erlang (1987)
- Haskell (1990)
- OCaml (1996)

Functional Style

Elements of functional ideas can be found in many modern languages (which would not usually be thought of as “functional languages”)

Functional Style

Elements of functional ideas can be found in many modern languages (which would not usually be thought of as “functional languages”)

E.g., Python, Scala, C#, JavaScript, Rust and many more

Functional Style

Elements of functional ideas can be found in many modern languages (which would not usually be thought of as “functional languages”)

E.g., Python, Scala, C#, JavaScript, Rust and many more

Java has just introduced first class functions (lambdas)

Functional Style

Elements of functional ideas can be found in many modern languages (which would not usually be thought of as “functional languages”)

E.g., Python, Scala, C#, JavaScript, Rust and many more

Java has just introduced first class functions (lambdas)

Though there have been many previous attempts to add functional style to Java, e.g., Pizza

Functional Style

Note that Lisp, OCaml and Haskell have OO systems as well as being functional

Functional Style

Note that Lisp, OCaml and Haskell have OO systems as well as being functional

In fact, their OO is much more powerful and flexible than Java or C++

Functional Style

Note that Lisp, OCaml and Haskell have OO systems as well as being functional

In fact, their OO is much more powerful and flexible than Java or C++

OO ideas were first developed in Lisp, before being added to C (giving C++ and Objective-C) or being part of the design of Java

Functional Style

Note that Lisp, OCaml and Haskell have OO systems as well as being functional

In fact, their OO is much more powerful and flexible than Java or C++

OO ideas were first developed in Lisp, before being added to C (giving C++ and Objective-C) or being part of the design of Java

Later we shall look at *metaboject protocols* where the behaviour of a OO system can be altered within the program

Functional Style

Note that Lisp, OCaml and Haskell have OO systems as well as being functional

In fact, their OO is much more powerful and flexible than Java or C++

OO ideas were first developed in Lisp, before being added to C (giving C++ and Objective-C) or being part of the design of Java

Later we shall look at *metaboject protocols* where the behaviour of a OO system can be altered within the program

If you don't like the way methods are chosen, or the way slots are accessed in an object, change it

Functional Style

We are going to look at two major examples of functional languages

Functional Style

We are going to look at two major examples of functional languages

That is, languages that encourage or force the functional style

Functional Style

We are going to look at two major examples of functional languages

That is, languages that encourage or force the functional style

- Lisp

Functional Style

We are going to look at two major examples of functional languages

That is, languages that encourage or force the functional style

- Lisp
- Haskell

Functional Style

We are going to look at two major examples of functional languages

That is, languages that encourage or force the functional style

- Lisp
- Haskell

Two languages that look and feel very different, but both embrace the functional idea

Lisp

- First appeared in 1959

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax
- So looks weird to those unused to it

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax
- So looks weird to those unused to it
- Originally procedural, but later discovered it was naturally functional

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax
- So looks weird to those unused to it
- Originally procedural, but later discovered it was naturally functional
- Both OO and functional ideas first developed in Lisp

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax
- So looks weird to those unused to it
- Originally procedural, but later discovered it was naturally functional
- Both OO and functional ideas first developed in Lisp
- Very flexible as a language, doesn't force any style

Lisp

- First appeared in 1959
- Has a deceptively simple but powerful syntax
- So looks weird to those unused to it
- Originally procedural, but later discovered it was naturally functional
- Both OO and functional ideas first developed in Lisp
- Very flexible as a language, doesn't force any style
- Really a *family* of languages

Haskell

- First appeared in 1990

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax
- So things don’t always do what you think

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax
- So things don't always do what you think
- Designed from scratch as functional: actually developed from earlier functional languages like Miranda, ML and SASL

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax
- So things don't always do what you think
- Designed from scratch as functional: actually developed from earlier functional languages like Miranda, ML and SASL
- Has a powerful OO subsystem

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax
- So things don't always do what you think
- Designed from scratch as functional: actually developed from earlier functional languages like Miranda, ML and SASL
- Has a powerful OO subsystem
- Forces the functional style

Haskell

- First appeared in 1990
- Has a deceptively familiar syntax
- So things don't always do what you think
- Designed from scratch as functional: actually developed from earlier functional languages like Miranda, ML and SASL
- Has a powerful OO subsystem
- Forces the functional style
- Standardised as "Haskell 98"

Functional Languages

There are also other important differences that will become clearer later

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*
- Haskell is *non-strict* and *lazy*

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*
- Haskell is *non-strict* and *lazy*

Strict and eager is what you are used to from other languages

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*
- Haskell is *non-strict* and *lazy*

Strict and eager is what you are used to from other languages

Non-strict and lazy are probably new to you

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*
- Haskell is *non-strict* and *lazy*

Strict and eager is what you are used to from other languages

Non-strict and lazy are probably new to you

So Lisp will look strange but act as you might expect

Functional Languages

There are also other important differences that will become clearer later

- Lisp is *strict* and *eager*
- Haskell is *non-strict* and *lazy*

Strict and eager is what you are used to from other languages

Non-strict and lazy are probably new to you

So Lisp will look strange but act as you might expect

And Haskell will look relatively normal but act quite weirdly

Lisp

We start with Lisp

Lisp

We start with Lisp

> *“Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”*

Eric Raymond, “How to Become a Hacker”

Lisp

Lisp is a good language to learn since

Lisp

Lisp is a good language to learn since

- it is the “simplest” functional language

Lisp

Lisp is a good language to learn since

- it is the “simplest” functional language
- it has historical importance

Lisp

Lisp is a good language to learn since

- it is the “simplest” functional language
- it has historical importance
- it looks very different from other languages you have learnt

Lisp

Lisp is a good language to learn since

- it is the “simplest” functional language
- it has historical importance
- it looks very different from other languages you have learnt

We shall spend more time on Lisp than Haskell as they share many ideas

Lisp

Lisp is a good language to learn since

- it is the “simplest” functional language
- it has historical importance
- it looks very different from other languages you have learnt

We shall spend more time on Lisp than Haskell as they share many ideas

By the time we get to Haskell we should be able to say “and such-and-such is just like Lisp” and just concentrate on their differences

Lisp

History

Lisp is a **lis**t **p**rocessing language

Lisp

History

Lisp is a **lis**t **p**rocessing language

One of the oldest high level languages, developed in 1956-1958 by John McCarthy

Lisp

History

Lisp is a **lis**t **p**rocessing language

One of the oldest high level languages, developed in 1956-1958 by John McCarthy

Lisp 1.0 first appeared in 1959

Lisp

History

Lisp is a **lis**t **p**rocessing language

One of the oldest high level languages, developed in 1956-1958 by John McCarthy

Lisp 1.0 first appeared in 1959

Only Fortran and Algol 58 are older

Lisp

History

Lisp is a **lis**t **p**rocessing language

One of the oldest high level languages, developed in 1956-1958 by John McCarthy

Lisp 1.0 first appeared in 1959

Only Fortran and Algol 58 are older

It is *symbolic processing* oriented, not numerical

Lisp

History

Lisp is a **lis**t **p**rocessing language

One of the oldest high level languages, developed in 1956-1958 by John McCarthy

Lisp 1.0 first appeared in 1959

Only Fortran and Algol 58 are older

It is *symbolic processing* oriented, not numerical

Not designed for numerical processing (Fortran did that), but manipulation of symbols

Lisp

History

It was supposed to be a computer realisation of the mathematical theory of *Lambda Calculus*

Lisp

History

It was supposed to be a computer realisation of the mathematical theory of *Lambda Calculus*

The Lambda Calculus is a way of describing computation in such a way you can prove things mathematically

Lisp

History

It was supposed to be a computer realisation of the mathematical theory of *Lambda Calculus*

The Lambda Calculus is a way of describing computation in such a way you can prove things mathematically

In some sense, like Turing machines are a model of computation, but very different looking

Lisp

History

It was supposed to be a computer realisation of the mathematical theory of *Lambda Calculus*

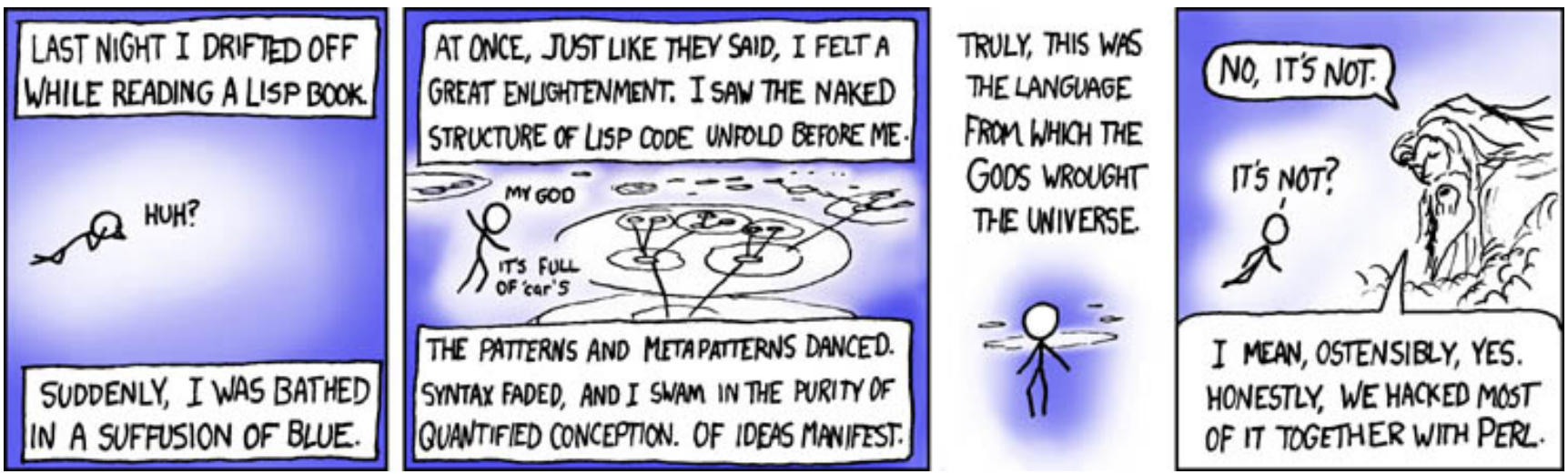
The Lambda Calculus is a way of describing computation in such a way you can prove things mathematically

In some sense, like Turing machines are a model of computation, but very different looking

We could do an entire unit on it

Lisp

History



http://xkcd.com/224/

Lisp

The basic datastructure is the *list*

Lisp

The basic datastructure is the *list*

Everything in Lisp is either

- a list
- or an *atom*

Lisp

The basic datastructure is the *list*

Everything in Lisp is either

- a list
- or an *atom*

`(1 x "hello")` is a list of three atoms: a number, a symbol and a string

Lisp

The basic datastructure is the *list*

Everything in Lisp is either

- a list
- or an *atom*

`(1 x "hello")` is a list of three atoms: a number, a symbol and a string

Symbols look like variables in other languages, but are objects in their own right

Lisp

There's not much you can do with symbols by default (in comparison, you can, e.g., add numbers or concatenate strings), they just *are*

Lisp

There's not much you can do with symbols by default (in comparison, you can, e.g., add numbers or concatenate strings), they just *are*

The original intent of Lisp was to make manipulation of symbols easy

Lisp

There's not much you can do with symbols by default (in comparison, you can, e.g., add numbers or concatenate strings), they just *are*

The original intent of Lisp was to make manipulation of symbols easy

That is, pushing symbols about, just like in mathematics

Lisp

`()` is the empty list

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

`(+ 1 2)` is another three item list: a symbol and two numbers

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

`(+ 1 2)` is another three item list: a symbol and two numbers

`((one 1) (two 2) (three 3) (four 4))` is a four item list, each of which is a list itself

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

`(+ 1 2)` is another three item list: a symbol and two numbers

`((one 1) (two 2) (three 3) (four 4))` is a four item list, each of which is a list itself

Lists can be nested arbitrarily

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

`(+ 1 2)` is another three item list: a symbol and two numbers

`((one 1) (two 2) (three 3) (four 4))` is a four item list, each of which is a list itself

Lists can be nested arbitrarily

`(((x 2) 3) ((x 1) 4) -1)` could be a representation of the polynomial $3x^2 + 4x - 1$

Lisp

`()` is the empty list

It is the only object that is both a list and an atom

`(+ 1 2)` is another three item list: a symbol and two numbers

`((one 1) (two 2) (three 3) (four 4))` is a four item list, each of which is a list itself

Lists can be nested arbitrarily

`(((x 2) 3) ((x 1) 4) -1)` could be a representation of the polynomial $3x^2 + 4x - 1$

Thus Lisp can easily be used to represent non-numeric data

Lisp

Modern Lisps have all kinds of other data types (vectors, characters, structures, general classes, etc.) but lists are the main idea we want here

Lisp

Modern Lisps have all kinds of other data types (vectors, characters, structures, general classes, etc.) but lists are the main idea we want here

The big thing about lists is that they are *dynamic*

Lisp

Modern Lisps have all kinds of other data types (vectors, characters, structures, general classes, etc.) but lists are the main idea we want here

The big thing about lists is that they are *dynamic*

They can grow and shrink as you add and remove elements from them

Lisp

Modern Lisps have all kinds of other data types (vectors, characters, structures, general classes, etc.) but lists are the main idea we want here

The big thing about lists is that they are *dynamic*

They can grow and shrink as you add and remove elements from them

In the 50s and 60s this was a novel and revolutionary idea: with Fortran you knew exactly how much memory a program would need just by looking at it

Lisp

Modern Lisps have all kinds of other data types (vectors, characters, structures, general classes, etc.) but lists are the main idea we want here

The big thing about lists is that they are *dynamic*

They can grow and shrink as you add and remove elements from them

In the 50s and 60s this was a novel and revolutionary idea: with Fortran you knew exactly how much memory a program would need just by looking at it

With a Lisp program you can't tell

Lisp

Syntax

Here is a bit of Lisp code that adds a pair of numbers

Lisp

Syntax

Here is a bit of Lisp code that adds a pair of numbers

```
(+ 1 2)
```

Lisp

Syntax

Here is a bit of Lisp code that adds a pair of numbers

```
(+ 1 2)
```

The syntax of Lisp is very simple: in other languages you might write `f(x,y)`

Lisp

Syntax

Here is a bit of Lisp code that adds a pair of numbers

```
(+ 1 2)
```

The syntax of Lisp is very simple: in other languages you might write `f(x,y)`

In Lisp you simplify this by dropping the comma and moving the parenthesis out: `(f x y)`

Lisp

Syntax

Here is a bit of Lisp code that adds a pair of numbers

```
(+ 1 2)
```

The syntax of Lisp is very simple: in other languages you might write `f(x,y)`

In Lisp you simplify this by dropping the comma and moving the parenthesis out: `(f x y)`

All functions are like this, even things like + that are treated specially by other languages

Lisp

Syntax

People complain about the syntax of Lisp saying it has too many parentheses

Lisp

Syntax

People complain about the syntax of Lisp saying it has too many parentheses

Lisp = Lots of Irritating Silly Parentheses

Lisp

Syntax

People complain about the syntax of Lisp saying it has too many parentheses

Lisp $=$ Lots of Irritating Silly Parentheses

But that's just because they have become used to the syntaxes of other languages: Lisp is actually simpler

Lisp

Syntax

People complain about the syntax of Lisp saying it has too many parentheses

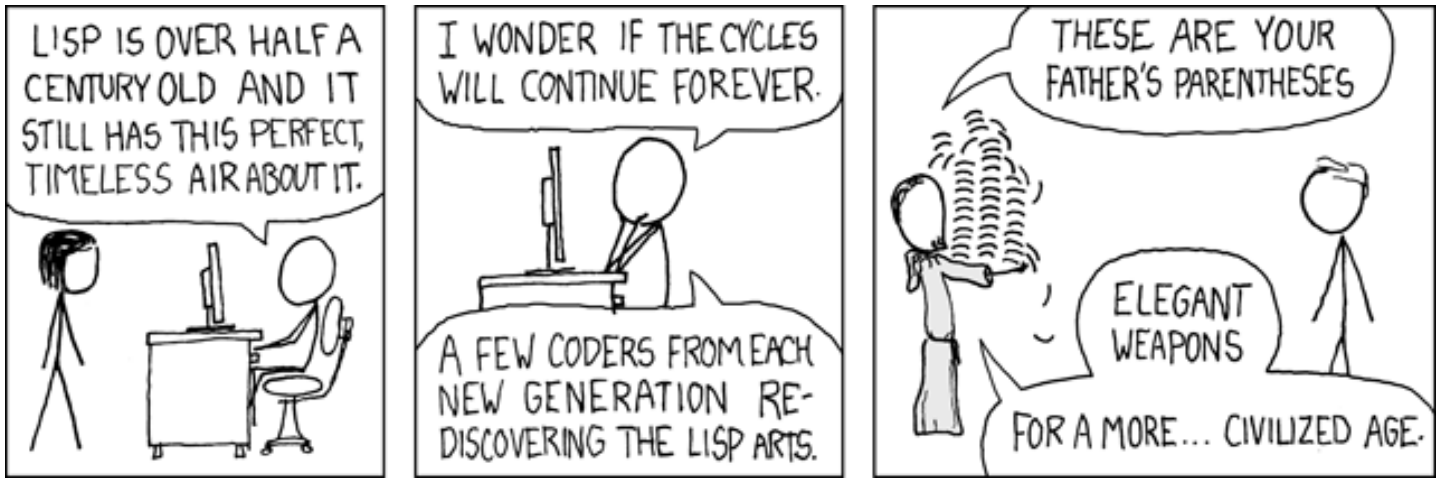
Lisp $=$ Lots of Irritating Silly Parentheses

But that's just because they have become used to the syntaxes of other languages: Lisp is actually simpler

And has exactly the right number of parentheses!

Lisp

Syntax



http://xkcd.com/297/

Lisp

Syntax

Like many things, it's a matter of practice and what you are used to

Lisp

Syntax

Like many things, it's a matter of practice and what you are used to

```
(+ (pow (sin x) 2) (pow (cos x) 2))
```

for $\sin^2 x + \cos^2 x$

Lisp

Syntax

Like many things, it's a matter of practice and what you are used to

```
(+ (pow (sin x) 2) (pow (cos x) 2))
```

for $\sin^2 x + \cos^2 x$

The *reason* for this syntax is very important