

CPS 305

Data Structures

Prof. Alex Ufkes

Topic 1: Course intro, Lisp intro

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 305 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Instructor



Alex Ufkes

aufkes@torontomu.ca

Class times:

Tuesday 1–4 (DCC 204)

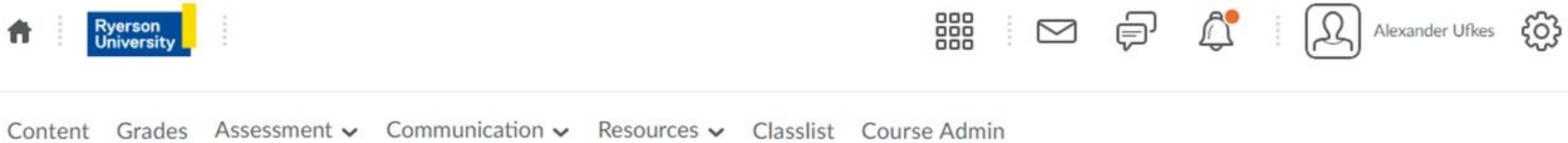
Lab times:

Check your schedule

When Contacting...

- E-mail – I check it often (**aufkes@torontomu.ca**)
- Please put CPS305 in the subject line
- Include your full name, use your TMU account

Course Administration



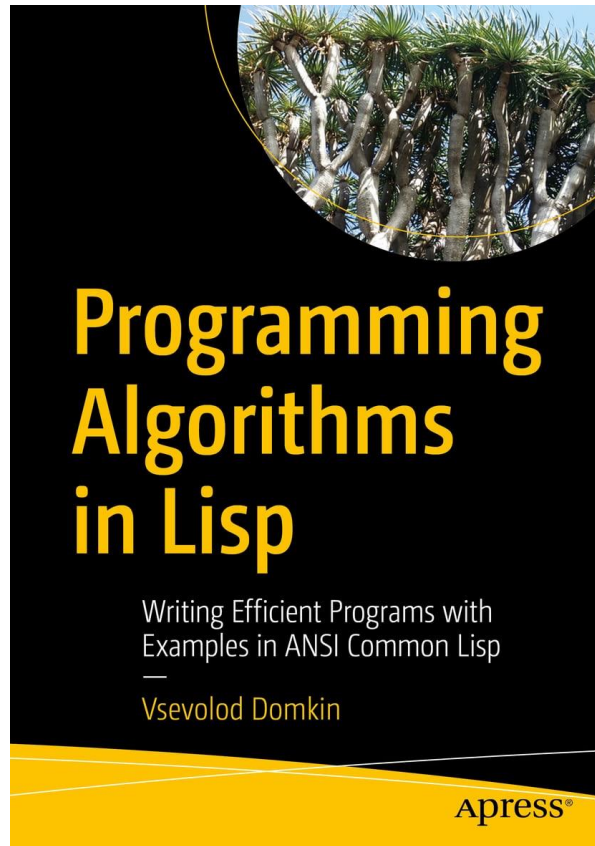
- Announcements related to this course will be made on D2L. Be sure to check regularly!
- Course outline, material, lecture slides, and grades will all be found on D2L.

Course Description

“Introduction to data structures and algorithms. Data structures covered will include stacks, queues, lists, trees, and graphs. Algorithm topics will include searching, sorting, hashing, algorithm design, greedy approaches, dynamic programming, recursion, and complexity analysis.”

Pretty much says it all.

Course Text



This book is available on Amazon for around \$50

Lecture slides will be posted every week, and there are lots of online resources.

Evaluation

Labs: 10%

- Eight practical labs, 1.25% each, starting in week 2.
- Writing Lisp programs. Completed alone.

Practical Test: 20%

- During lab period in week of July 14.

Midterm: 30%

- During class in week of June 16. Pen and paper.

Final Exam: 40%

- During final scheduled lecture. Pen and paper.

Regarding Deadlines

From the course outline:

Late Submissions

Late lab submissions will be penalized at a rate of $3^n\%$, where n is the number of days late. One day late is a 3% penalty, two days 9%, three days 27%, four days 81%, and five or more days receive zero.

- The late penalty starts small, but it ramps up quickly.

Any Questions?



Today

- Course motivation
- Data structure intro
- Lisp intro

Data Structures

A collection of data values, the relationships between them, and the operations that can be applied to them.

Built from primitive types

- A C-style array is a data structure
 - A simple sequence of primitives
-
- The efficiency of a data structure cannot be analyzed separately from the operations applied to them.
 - ***Accessing*** elements in a simple array is very efficient, but ***prepending*** elements is costly.

Data Structures

*Accessing elements is efficient, **prepending** elements is costly*

Accessing: Constant time

- Address of first element + index * sizeof(type)
- $a[5] = a + 5 * \text{sizeof}(\text{int})$



i	0	1	2	3	4	5	6	7	8	9	10	11	12
a[i]	9	4	7	12	14	19	0	1	2	8	3	7	7



Prepending: Linear time

- Every element must shift to the right by one

Data Structures

*Accessing elements is efficient, **prepending** elements is costly*

Picking the right data structure is important!

- Which operations are being performed most often?
- Do you require duplication? Order-based operations?

i	0	1	2	3	4	5	6	7	8	9	10	11	12
a[i]	9	4	7	12	14	19	0	1	2	8	3	7	7

Foundations of Computer Science

A deep understanding of data structures is one thing that separates computer programmers from computer scientists.



Consider a Python set:

- You know what it is, how to use it
- But **why** are some operations efficient, while others aren't?
- **Why** can't we have duplicate values?
- A deeper understanding lets us use built-in types more effectively.
- It also lets us design/adapt our own!

Foundations of Computer Science

Foundation is important!



Python Sets VS Lists

- Many functions/methods are the same for both.
- However! Some might be linear time over a list, constant time over a set.
- Very, **very** big difference when data size scales up.
- Pick the right ADT for the job!

ADT?

Data structures implement *Abstract Data Types* (ADTs)



This is a queue.

- It is First-in First-out (FIFO).
- FIFO is a good choice – what if drive-throughs were LIFO?
- = Stack

Data Structures VS ADTs

A **Queue** is an ADT.

- Since there's no such thing as a Queue primitive type, we must implement it ourselves somehow (or let the programming language do it).
- We can use an **array** to store the **data** of the queue and then implement methods or functions for operating on the array in a manner consistent with defined Queue behavior.

ADT is the **concept**; the data structure is the **implementation**.

Data Structures VS Algorithms

- This course is called “Data Structures”, but we will also study algorithms that operate on these data structures.
- These two concepts are inextricably linked
- A data structure without algorithms to operate on it would not be terribly useful.

For example:

- Lists/arrays, trees, and graphs should all be searchable.
- Search algorithm will differ depending on the ADT.

Algorithm Characteristics

CPS109:

- Steps must be in the correct order
- Operations must be unambiguous
- Operations must be feasible
- A result must be produced ...
- ... in a finite amount of time

Algorithms

al·go·rithm

/ˈalgəˌrɪθəm/

noun

a process or set of rules to be followed in calculations or other problem-solving operations,
especially by a computer.

"a basic algorithm for division"

In short, a procedure to solve a problem

Algorithms

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions, usually intended to accomplish a specific purpose.

- Jeff Erickson

<http://jeffe.cs.illinois.edu/teaching/algorithms/>

Data Structure & Algorithm Analysis

In computing, as in life, for every problem there exists many, many solutions

(Not all of which are created equal)



“Solution”



Problem



Solution

“...a procedure to solve a problem”

In computing, as in life, for every problem there exists many, many solutions

(Not all of which are created equal)



By what metric(s) do we declare one algorithm or data structure *better* than another?

Data Structure & Algorithm Analysis

We want to be able to say:

“Algorithm A is ***better*** than algorithm B”

What’s better about it?

- Efficiency? With respect to what?
- CPU cycles, memory footprint, code size, code simplicity?
- What do we care about most?

Binary search is “faster” than linear search, but only works on sorted lists

Data Structure & Algorithm Analysis

We want to be able to say:

“Algorithm A is ***faster*** than algorithm B”

Faster is more concrete than ***better***, but still isn't precise.

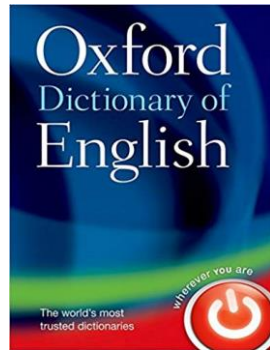
Faster under what circumstances?

- Best vs worst case cost, nature of the input matters.
- We might get lucky when searching and find the element in the first position!

Data Structure & Algorithm Analysis

Qualitative VS. Quantitative

“Better” and “faster” are ***qualitative*** comparisons, much like “prettier” and “tastes better”



Qualitative:

- Relating to, measuring, or measured by the quality of something rather than its quantity.
- Often contrasted with *quantitative*

Data Structure & Algorithm Analysis

Qualitative VS. Quantitative

Engineering design ***cannot*** be guided by qualitative comparisons.

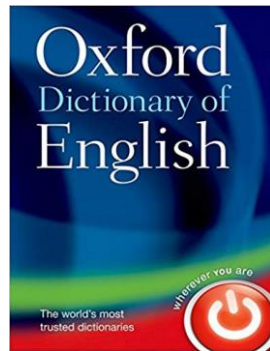
Algorithm A might be beneficial in some circumstances,
algorithm B might be beneficial in others.

Our decision to use A over B cannot be based on A being
“better” (or “simpler”, “more elegant”, etc.)

Data Structure & Algorithm Analysis

Qualitative VS. Quantitative

Instead, we will examine ***quantitative*** metrics for describing algorithms

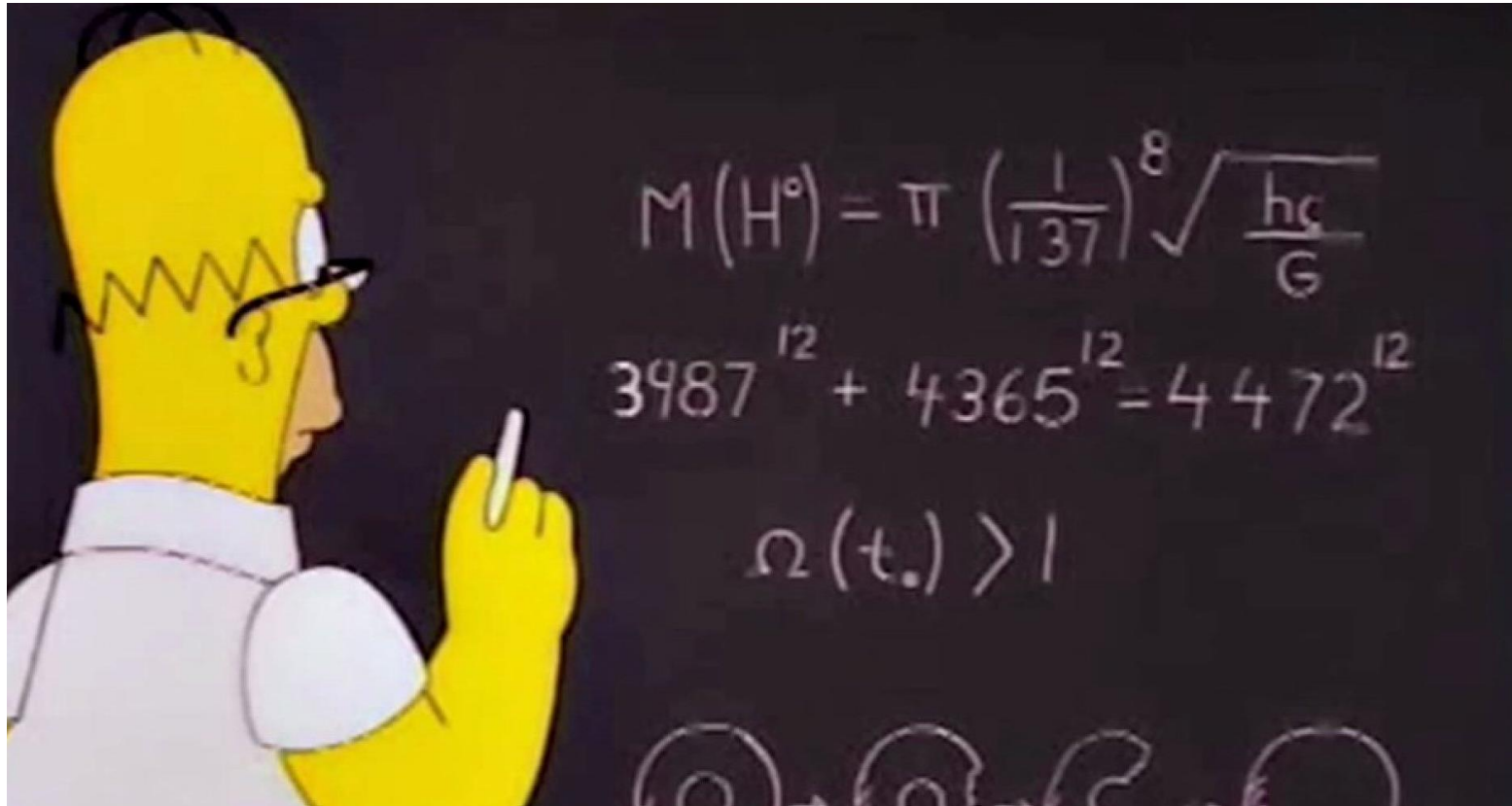


Quantitative:

- Relating to, measuring, or measured by the quantity of something rather than its quality.
- Often contrasted with qualitative

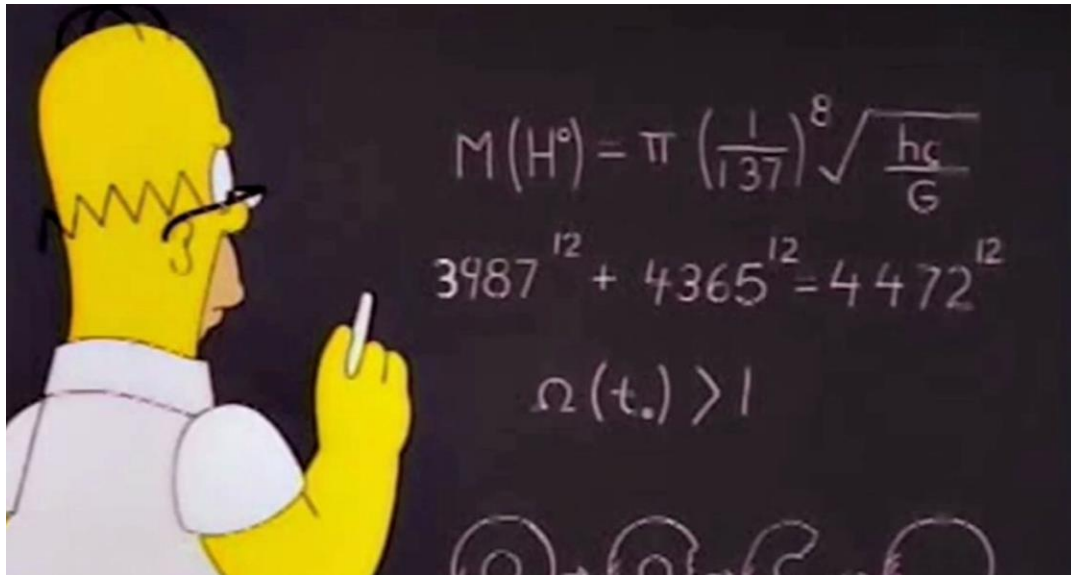
Data Structure & Algorithm Analysis

How do we describe algorithms and data structures quantitatively?



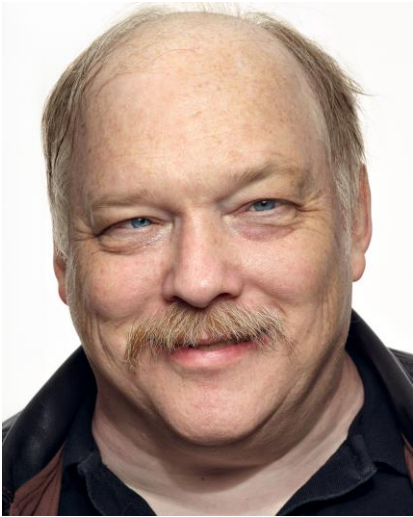
Data Structure & Algorithm Analysis

How do we describe algorithms and data structures quantitatively?

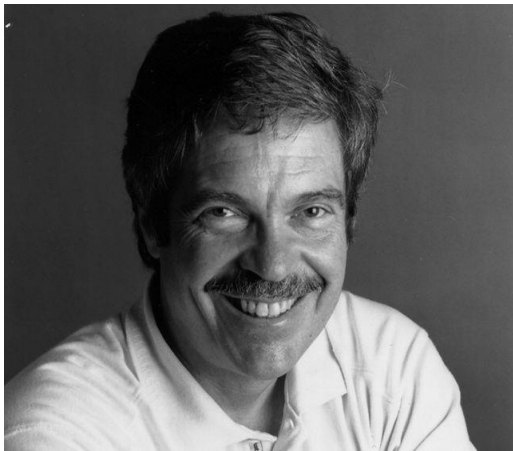


- Next class, we will lay the foundation for doing this quantitatively.
- Time/space complexity analysis, Big-O notation, etc.
- Before we get there, we'll have a crash course in the programming language used in this course.





"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



"Lisp isn't a language, it's a building material."
— Alan Kay

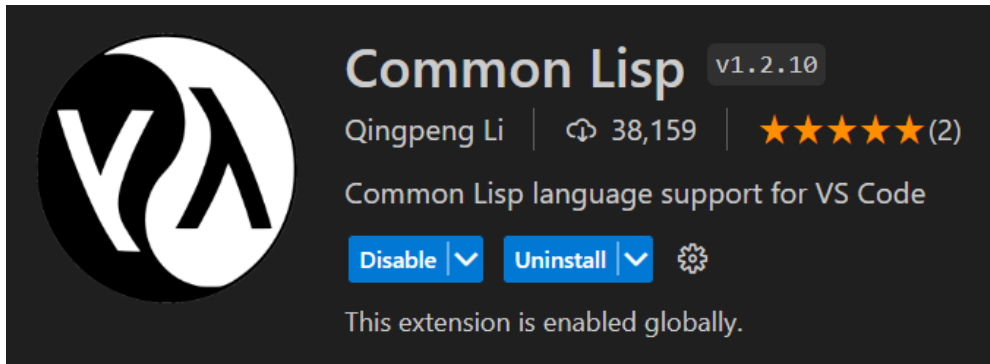
Common Lisp: History



- Developed by John McCarthy in 1958 (predated only by Fortran!)
- Pioneered many CS concepts: Tree ADTs, recursion, higher-order functions, etc.
- **LISP** = **LI**St **P**rocessor. Linked Lists are one of Lisp's major data structures.
- Many dialects of Lisp today: Common Lisp, Clojure, Scheme, etc.
- We will use ***Common Lisp***

Common Lisp: IDE

VSCode + Common Lisp extension



Steel Bank Common Lisp (SBCL) compiler

	X86	AMD64	PPC	PPC64	PPC64le	SPARC	MIPSbe	MIPSle	ARMel	ARMhf	ARM64	RISC-V 32	RISC-V 64
Linux	1.4.3	2.4.4 <i>newest</i>	1.2.7		1.5.8	1.0.28	1.0.23	1.0.28	1.2.7	2.3.3	1.4.2		
macOS (Darwin)	1.1.6	2.2.9									2.4.0		
Solaris	1.2.7	1.2.7				2.0.4							
FreeBSD	1.2.7	1.2.7									2.2.0		
NetBSD	1.0.22	1.2.7	1.0.23										
OpenBSD	2.0.5	2.0.5	2.0.5						2.0.5		2.0.5		
DragonFly BSD		1.2.7											
Debian GNU/kFreeBSD	1.2.7	1.2.7											
Windows	2.3.2	2.4.4 <i>newest</i>											

<https://www.sbcl.org/platform-table.html>

Common Lisp: Resources

Common Lisp Cookbook:

<https://lispcookbook.github.io/cl-cookbook/>

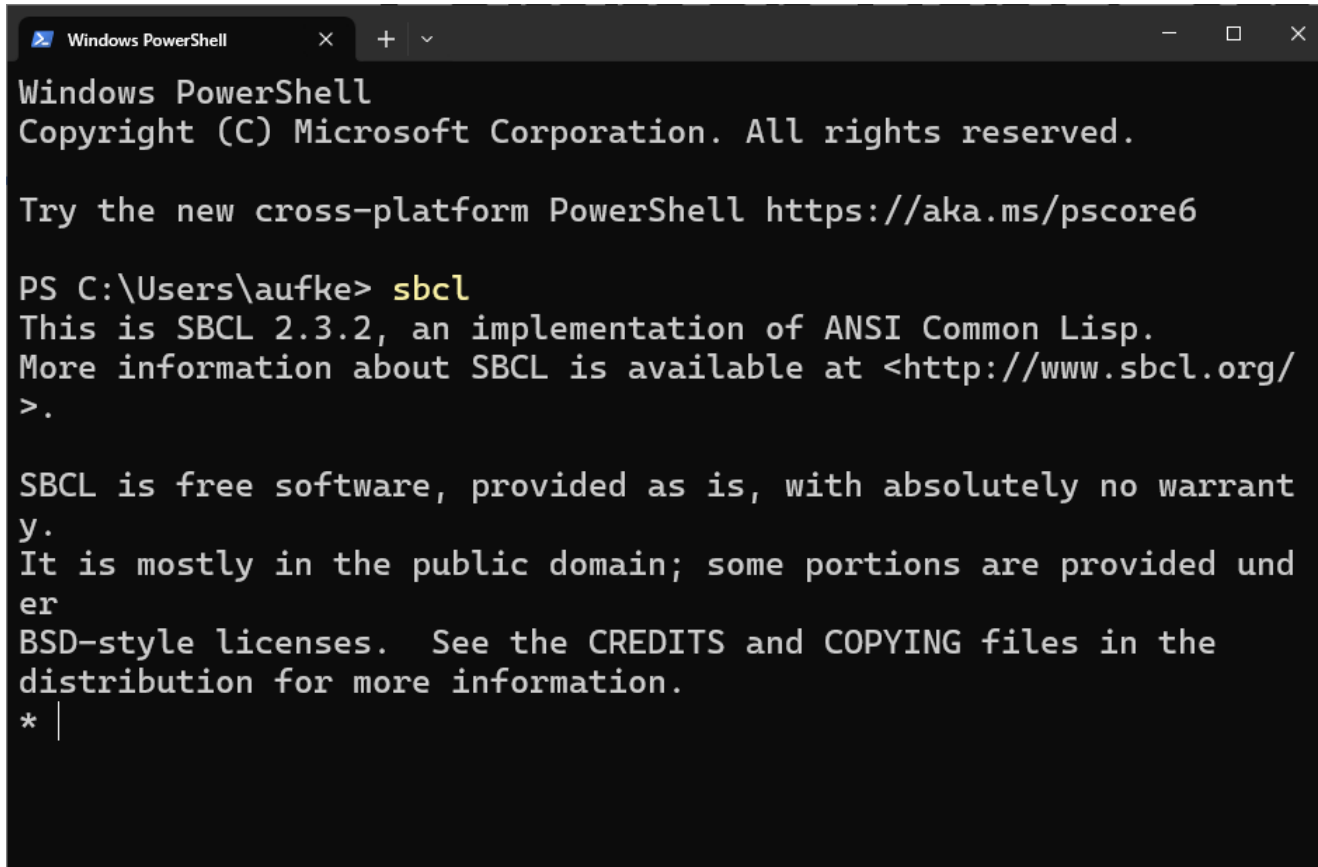
CL HyperSpec (complete documentation):

<http://clhs.lisp.se/Front/index.htm>

Lisp Tutorialspoint

<https://www.tutorialspoint.com/lisp/index.htm>

Common Lisp: Getting Started



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke> sbcl
This is SBCL 2.3.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warrant
y.
It is mostly in the public domain; some portions are provided und
er
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
* |
```

- Install SBCL, type 'sbcl' into a terminal.
- This opens a REPL (Read Eval Print Loop)
- You are now in conversation with Lisp
- Hello, world?

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke> sbcl
This is SBCL 2.3.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/
>.

SBCL is free software, provided as is, with absolutely no warrant
y.
It is mostly in the public domain; some portions are provided und
er
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
* (format t "Hello, world!")
Hello, world!
NIL
* |
```

Common Lisp: Getting Started

This is a Lisp ***Form*** (expression):

- **format**: function for printing
- **t**: Print destination (stdout)

A diagram illustrating a Common Lisp REPL session. A green bracket above the code block groups the entire expression. A blue arrow points from a text box to the string "Hello, world!". A red arrow points from a text box to the "NIL" output. The code block contains the following text:

```
* (format t "Hello, world!")  
Hello, world!  
NIL  
* |
```

"Hello, world!" is printed:

NIL is returned (and echoed)

Common Lisp: Getting Started

This is a Lisp ***Form***, defined by parentheses

- **format**: function for printing
- **t**: When NIL, string is returned (not printed)

```
* (format t "Hello, world!")
Hello, world!
NIL
* (format NIL "Hello, world!")
"Hello, world!"
* |
```

Common Lisp: Scripts

We can interact with Lisp via REPL, but also via script:

```
λ hello-world.lisp ×
λ hello-world.lisp
1 ; Run script, do not launch REPL:
2 ; sbcl --script hello-world.lisp
3 ; Run script, then launch REPL:
4 ; sbcl --load hello-world.lisp
5 ; Exit REPL:
6 ; (SB-EXT:EXIT) to exit
7
8 (format t "Hello, world!")
9
10 |
```

(QUIT)
also exits

Common Lisp: Scripts

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> sbcl --script hello-world.lisp
Hello, world!
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> sbcl --load hello-world.lisp
This is SBCL 2.3.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
Hello, world!
* (SB-EXT:EXIT)
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> |
```

```
hello-world.lisp x
λ hello-world.lisp
1 ; Run script, do not launch REPL:
2 ; sbcl --script hello-world.lisp
3 ; Run script, then launch REPL:
4 ; sbcl --load hello-world.lisp
5 ; Exit REPL:
6 ; (SB-EXT:EXIT) to exit
7
8 (format t "Hello, world!")
9
10 |
```



Forms

Lisp Forms: Literals

- The concept of a “Form” is central in Lisp.
- They are **expressions** that can be evaluated.
- We just saw one example:

```
* (format t "Hello, world!")  
Hello, world!  
NIL  
* |
```

Lisp Forms: Literals

Literals are forms that evaluate to themselves:

```
* "Hello, world!"  
"Hello, world!"  
* 10  
10  
* 10.0  
10.0  
* 20/2  
10  
* #xA  
10  
* #b1010  
10  
* |
```

Strings, double quoted
sequence of characters

Not division! This is a ratio literal.

Hex and binary literals

Lisp Forms: Literals

Character literals:

```
* #\A
#\A
* #\b
#\b
* #\"
#\ "
* |
```

- They don't seamlessly convert to ASCII or Unicode values as in many other languages.
- We can't treat them the same way as numbers.
- Can't use numeric comparison on characters (<, >, etc.)
- `#\` followed by a character

Lisp Forms: write + newlines

λ newlines.lisp

```
1
2 (write 7)
3 (terpri)      ; terminate print (add newline)
4 (terpri)      ; terminate print (add newline)
5 (write 21/7)
6 (fresh-line) ; go to next line, if not already at the start of a line
7 (fresh-line) ; go to next line, if not already at the start of a line
8 (write 7.0)
9
10
```

Windows PowerShell

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

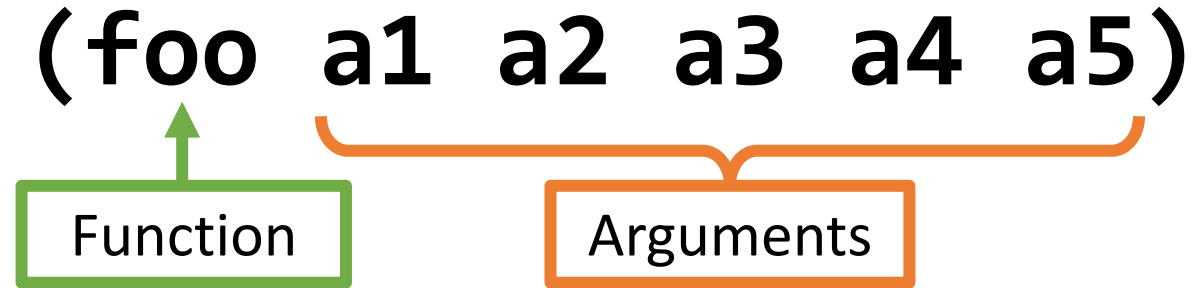
Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> sbcl --script newlines.lisp
7

3
7.0

PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> |

Lisp Forms: Function Application



- If the first element is **NOT** a special form, apply as function to arguments.
- If it **IS** a special form, it is treated differently (more later)

Lisp Forms: Function Application

(foo a1 a2 a3 a4 a5)

```
* (+ 7 2)
9
* (- 7 2)
5
* (* 18 4)
72
* (/ 1 3)
1/3
* (/ 6 2)
3
* |
```

- Arithmetic performed as function application
- Lisp uses *prefix* (as opposed to *infix*) notation
- (+ 2 3) VS (2 + 3)

Lisp Forms: Function Application

$$(2 + \sqrt{4} * 6) * (3 + 5 + 7)$$

```
* (* (+ 2 (* (sqrt 4) 6)) (+ 3 5 7))  
210.0
```

$$\frac{2 - (3 - (6 + \frac{4}{5}))}{3 * (6 - 2)}$$

```
* (/ (- 2 (- 3 (+ 6 (/ 4 5)))) (* 3 (- 6 2)))  
29/60
```

Remember this is the prompt,
not part of a form...

Lisp Forms: Special Forms

- *If the first element is **NOT** a special form, apply function to arguments.*
- *If it **IS** a special form, it is treated differently (more later)*

There are **25** special forms, each having their own special evaluation rules

We'll start with the most common:
QUOTE, IF, LAMBDA, LET, DEFUN

Lisp Forms: Special Forms

QUOTE

IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Return a Lisp form as data, rather than evaluating it:

```
* (+ 2 3)
```

```
5
```

```
* (quote (+ 2 3))
```

```
(+ 2 3)
```

```
* '(+ 2 3)
```

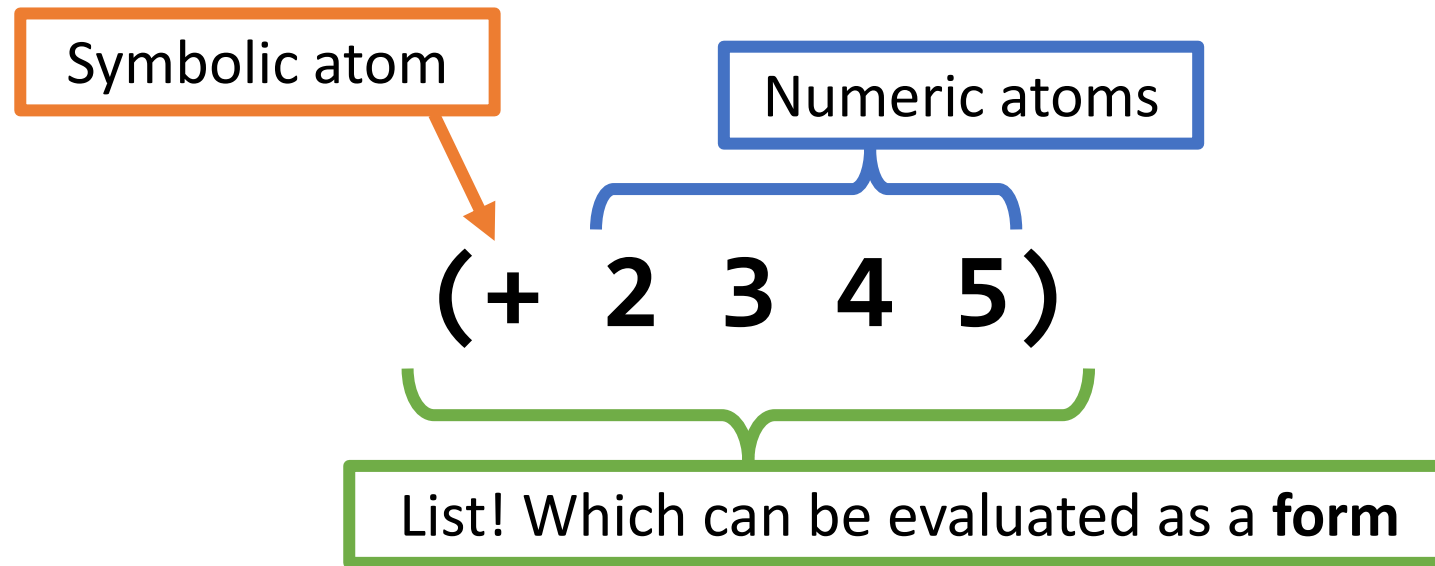
```
(+ 2 3)
```

Can use **quote** or **'**

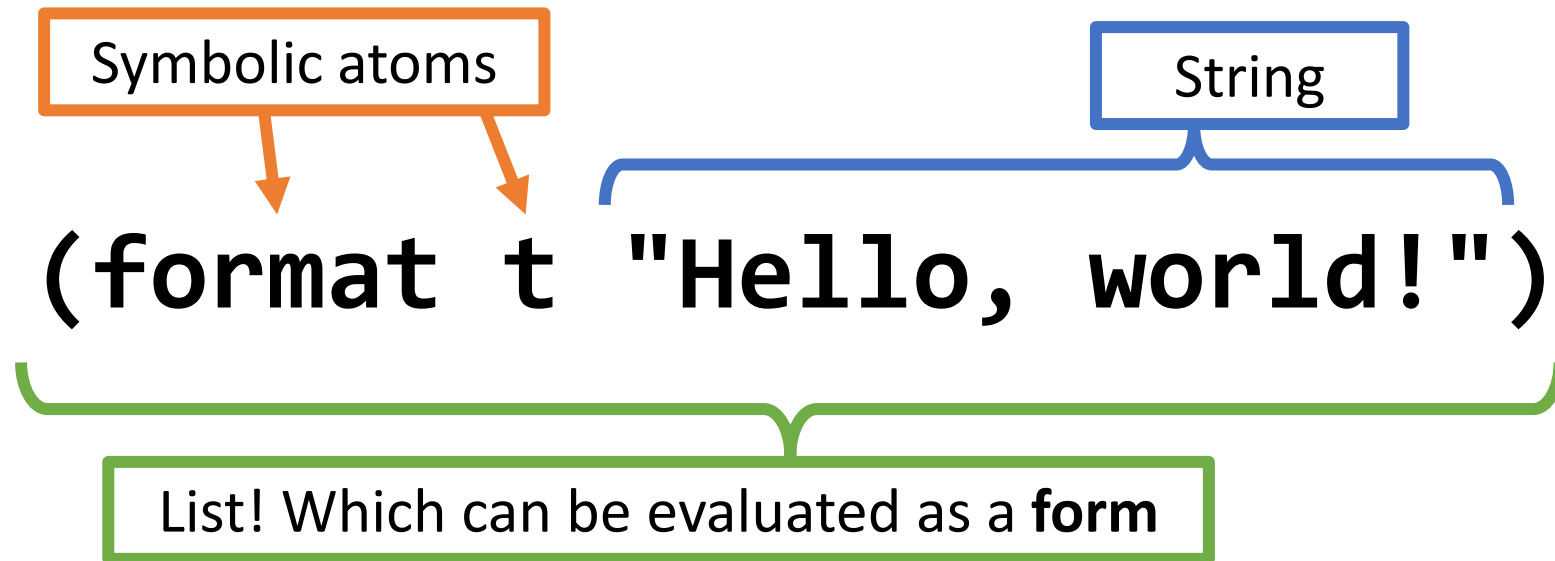


Lisp Forms: As Data?

- Forms are *lists* of *atoms*
- Symbolic atoms are made up of letters, numbers, characters, etc.
- Along with *strings*, these are the basic building blocks of Lisp programs



Lisp Forms: As Data?



```
Windows PowerShell
* (1 2 3 4)
; in: 1 2
; (1 2 3 4)
;
; caught ERROR:
; illegal function call
;
; compilation unit finished
; caught 1 ERROR condition
```

- List containing 1, 2, 3, 4
- REPL evaluates as a form
- Tries to call function named 1

```
debugger invoked on a SB-INT:COMPILED-PROGRAM-
#<THREAD "main thread" RUNNING {234E0001}>:
  Execution of a form compiled with errors.
Form:
  (1 2 3 4)
Compile-time error:
  illegal function call

Type HELP for debugger help, or (SB-EXT:EXIT)

restarts (invokable by number or by possibly-a
0: [ABORT] Exit debugger, returning to top l

((LAMBDA ()))
  source: (1 2 3 4)
0] |
```

```
Windows PowerShell

restarts (invokable by number or by po
0: [ABORT] Exit debugger, returning

((LAMBDA ()))
  source: (1 2 3 4)
0] abort

* '(1 2 3 4)
(1 2 3 4)
* |
```

Quote to return
form as a list

Lisp Forms: Special Forms

QUOTE

IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Return a Lisp form as data, rather than evaluating it:

```
* (+ 2 3)
```

```
5
```

```
* (quote (+ 2 3))
```

```
(+ 2 3)
```

```
* '(+ 2 3)
```

```
(+ 2 3)
```

Thus:

If we simply want a list containing atoms, we can quote it.
Otherwise, it evaluates as a form.

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

First we need logical operators (and, or, not):

- They short circuit, Return truthy/falsy.
- **NIL** and **()** are false, everything else is **T** (true)

* (and T T NIL T)

NIL

* (and T T T 42)

42

* (or NIL 0 42)

0

* (or NIL NIL NIL NIL)

NIL

42 decided the result,
it gets returned

0 decided the result, it gets
returned (yes, 0 is true!)

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

We also need comparison operators:

- With numbers, these are simple enough
- Character comparison uses different functions

* (< 1 2)

T

* (> 1 2)

NIL

* (>= 2 2)

T

* (>= 2 3)

NIL

* (= 2 3)

NIL

* (/= 2 3)

T

* (min 2 4 6 5 3)

2

* (max 2 4 6 5 3)

6

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

Fancier, nesting forms:

```
* (and 0 1 nil (+ 3 4) 3)
```

```
NIL
```

```
* (and 0 1 2 (* 3 3))
```

```
9
```

```
* (or 10 (* 3 3) 2 1)
```

```
10
```

```
* (or (> 5 6) (< 3 4) (= 1 2))
```

```
T
```

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

Incredible! Tests if args are increasing/decreasing

- Applies function to every pair of adjacent arguments

*** (< 1 2 3 4 5)**

T

*** (< 1 2 3 2 1)**

NIL

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

(if testForm thenForm [elseForm])

Condition

True form

False form

Notation: [] indicates optional (single branch allowed)

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

(if testForm thenForm [elseForm])

*** (if (> 6 3) (+ 5 5))**

10

*** (if (> 1 3) (+ 5 5))**

NIL

*** (if (> 6 3) (+ 5 5) (* 5 5))**

10

*** (if (> 1 3) (+ 5 5) (* 5 5))**

25

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

(if testForm thenForm [elseForm])

```
* (if (> 6 3) (format t "Hello") (format t "World"))
```

```
Hello
```

```
NIL
```

```
* (if (> 1 3) (format t "Hello") (format t "World"))
```

```
World
```

```
NIL
```

*Recall: format returned NIL, Hello/World
printed to stdout as side effect.*

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

Lambdas are *first-class* functions:

- First class? Can be passed as values to another function.
- (Almost) every language has them, in one form or another.
- Lambdas, closures, blocks, anonymous functions, etc.
- They don't need names, and as such have a literal syntax.

Parameter list

Function behavior

* (lambda (x) (* x x))

#<FUNCTION (LAMBDA (X)) {235504DD}>

Lisp Forms: Special Forms

QUOTE

IF

LAMBDA

LET

DEFUN

CASE

DEFVAR

```
* ((lambda (x) (* x x)) 9)
```

```
81
```

```
* ((lambda (x) (* x x)) 5)
```

```
25
```

- Weird way to use lambdas, but we can use them as the first element of a regular old form.
- We'll see better uses later on.

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Temporary variables in an expression:

(let ((x_1 e_1) (x_2 e_2) ... (x_i e_i)) e)

1. All expressions e_i bound to variables x_i
2. Expression e is evaluated, may refer to any x_i

*** (let ((a 1) (b 2) (c 3)) (+ a b c))**
6

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

e_i can be any form, or series of nested forms:

(let ((x_1 e_1) (x_2 e_2) ... (x_i e_i)) e)

*** (let ((a (+ 4 2)) (b (* 4 5))) (/ a b))
3/10**

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Temporary variables in an expression:

(let ((x₁ e₁) (x₂ e₂) ... (x_i e_i)) e)

1. All expressions **e_i** bound to variables **x_i**
2. Expression **e** is evaluated, may refer to any **x_i**

*** (let ((a 1) (b a) (c b)) (+ a b c))
--> ERROR!**

Temporary variables are bound in parallel, not in series

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

```
* (let ((a 1) (b a) (c b)) (+ a b c))  
--> ERROR!
```

Temporary variables are bound in parallel, not in series

Use let* instead to bind in series:

```
* (let* ((a 1) (b a) (c b)) (+ a b c))  
3
```

```
λ arithmetic.lisp  λ special-forms.lisp ×  λ hello-world.lisp
λ special-forms.lisp > ...
1  (write
2    (let* ( (a 2)
3            (b 4)
4            (c -1)
5            (delta (- (* b b) (* 4 a c)))
6            (res (sqrt delta)) )
7            (/ (+ (- b) res) (* 2 a)))
8    )
9  )
10
11
```

In a script: Indent as you like to increase readability

```
sbc1 --script special-forms.lisp
0.22474492
```

- There are other more elaborate variants of let*
- We'll explore them later, as needed.

Pop Quiz! What is the Result?

* (+ 5 3 4)

12

* (- 9 1)

8

* (+ (* 2 4) (- 4 6))

6

* (lambda (x y) (+ x y))

#<FUNCTION (LAMBDA (X Y)) {2355023D}>

* ((lambda (x y) (+ x y)) (+ 2 1) 4)

7

* (let ((a 4) (b 5) (c 30)) (+ (* a b) c))

50

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

let binds temporary variables, **defun** creates a function in memory (not temporary!)

General format:

(defun name (params) "doc" (behavior))

Example:

**(defun average (x y)
 "Returns the average of x and y"
 (/ (+ x y) 2.0))**

```
λ arithmetic.lisp  λ special-forms.lisp  λ functions.lisp ×
λ functions.lisp > ...
1  (defun average (x y)
2    "Returns the average of x and y"
3    (/ (+ x y) 2.0))
4
```

```
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> sbcl --load functions.lisp
This is SBCL 2.3.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
* (average 2 4)
3.0
* (average 5 9)
7.0
* (average 5 8)
6.5
* 
```

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Example: Function to test if argument is odd

```
(defun isodd (x)
  "Returns T if x is odd, NIL otherwise"
  (= (rem x 2) 1))
```

Example: Function to add 1 to number, if odd:

```
(defun makeeven (x)
  "Returns x if x is odd, x+1 otherwise"
  (if (isodd x) (+ x 1) x))
```

λ arithmetic.lisp λ special-forms.lisp λ functions.lisp ×

λ functions.lisp > ...

```
1 (defun average (x y)
2   "Returns the average of x and y"
3   (/ (+ x y) 2.0))
4
5 (defun isodd (x)
6   "Returns T if x is odd, NIL otherwise"
7   (= (rem x 2) 1))
8
9 (defun makeeven (x)
10  "Returns x if x is odd, x+1 otherwise"
11  (if (isodd x) (+ x 1) x))
12
13
```

```
PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp examples> sbcl --load functions.lisp
This is SBCL 2.3.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.
```

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.

```
* (isodd 1)
T
* (isodd 2)
NIL
* (isodd 3)
T
* (isodd 4)
NIL
* (makeeven 1)
2
* (makeeven 2)
2
* (makeeven 3)
4
* (makeeven 4)
4
*
```

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

```
(defun binary-to-bool (val)
  (case val (0 NIL) (1 T)))
```

Value to test

If val is 0,
return NIL

If val is 1,
return T

```
* (defun binary-to-bool (val) (case val (0 NIL) (1 T)))
BINARY-TO-BOOL
* (binary-to-bool 0)
NIL
* (binary-to-bool 1)
T
```

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

```
(defun letter-grade (numeric-grade)
  "Convert numeric grade to letter grade"
  (case (floor numeric-grade 10)
    (10 "A")
    (9 "A")
    (8 "A")
    (7 "B")
    (6 "C")
    (5 "D")
    (otherwise "F")))
```

λ functions.lisp > ...

```

1  (defun average (x y)
2    "Returns the average of x and y"
3    (/ (+ x y) 2.0))
4
5  (defun isodd (x)
6    "Returns T if x is odd, NIL otherwise"
7    (= (rem x 2) 1))
8
9  (defun makeeven (x)
10   "Returns x if x is odd, x+1 otherwise"
11   (if (isodd x) (+ x 1) x))
12
13  (defun binary-to-bool (val) (case val (0 NIL) (1 T)))
14
15  (defun letter-grade (numeric-grade)
16    "Convert numeric grade to letter grade"
17    (case (floor numeric-grade 10)
18      (10 "A")
19      (9 "A")
20      (8 "A")
21      (7 "B")
22      (6 "C")
23      (5 "D")
24      (otherwise "F")))
25
26

```

```

PS C:\Users\aufke\Google Drive\Teaching\CPS 305\Lisp
This is SBCL 2.3.2, an implementation of ANSI Common
More information about SBCL is available at <http://v

```

SBCL is free software, provided as is, with absolute
It is mostly in the public domain; some portions are
BSD-style licenses. See the CREDITS and COPYING file
distribution for more information.

```

* (letter-grade 100)
"A"
* (letter-grade 97)
"A"
* (letter-grade 84)
"A"
* (letter-grade 79)
"B"
* (letter-grade 67)
"C"
* (letter-grade 55)
"D"
* (letter-grade 43)
"F"
*

```

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

```
(defun mypair (x y msg)
  (case msg
    (sum (+ x y))
    (diff (abs (- x y)))
    (dist (sqrt (+ (* x x) (* y y))))
    (print (pprint x) (pprint y))))
```

```
* (mypair 3 2 'print)
3
2
* (mypair 3 2 'sum)
5
* (mypair 3 2 'diff)
1
```


Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

Let's play a game: We think of a number, and Lisp tries to guess it!

1. Define lower and upper bounds
2. Choose a number between these bounds
3. If the player says smaller, lower the upper bound
4. If the player says larger, raise the lower bound

The game will be played in the REPL, we'll use
global variables for the bounds.

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

We'll use global variables for the bounds.

```
* (defvar *lowerb* 1)
*LOWERB*
* (defvar *upperb* 100)
*UPPERB*
* *lowerb*
1
* *upperb*
100
```

- Asterisks, or “earmuffs”, are part of the variable name.
- This is convention for global variables in Lisp.

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE

DEFVAR

- Lisp will use binary search to guess the number.
- Pick halfway point, raise or lower bounds accordingly.
- Use the **ash** function to efficiently divide/multiply by 2:

```
* (ash 5 -1)
2
* (ash 6 -1)
3
* (ash 6 1)
12
```

Note: *Bit-shifting is MUCH faster than division when the 2nd operand is a power of 2.*

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE
DEFVAR

We've declared two global variables, but Lisp will have to update (mutate) them as it makes unsuccessful guesses.

Beware global mutation:

- We can do it in Lisp, but it's generally to be avoided.
- Mutating variables in the global scope can have side effects elsewhere in your code.
- Always mutate responsibly!

Variables bound using let exist in the scope of their own form, so there's no danger of side effects.

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE

DEFVAR

We've declared two global variables, but Lisp will have to update (mutate) them as it makes unsuccessful guesses.

Use **setf**:

```
* (defvar *lowerb* 1)
*LOWERB*
* (defvar *upperb* 100)
*UPPERB*
* *lowerb*
1
* *upperb*
100
```

```
* (setf *lowerb* 25)
25
* (setf *upperb* 50)
50
* *lowerb*
25
* *upperb*
50
```

Lisp Forms: Special Forms

QUOTE
IF
LAMBDA
LET
DEFUN
CASE

DEFVAR

```
(defun guess-my-number ()  
  (ash (+ *upperb* *lowerb*) -1))
```

```
(defun smaller ()  
  (setf *upperb* 1- (guess-my-number)))  
  (guess-my-number))
```

```
(defun bigger ()  
  (setf *lowerb* 1+ (guess-my-number)))  
  (guess-my-number))
```

Functions for increment/decrement

λ guess-number.lisp > ...

```
1 (defvar *lowerb* 1)
2 (defvar *upperb* 100)
3
4 (defun guess ()
5   "Returns the integer mean of *upperb* and *lowerb*"
6   (ash (+ *upperb* *lowerb*) -1))
7
8 (defun smaller ()
9   (setf *upperb* (1- (guess)))
10  (guess))
11
12 (defun bigger ()
13   (setf *lowerb* (1+ (guess)))
14   (guess))
15
16 (defun start-over ()
17   (setf *lowerb* 1)
18   (setf *upperb* 100)
19   (guess))
20
21
```

Pick a number in your head:

- Let's say... 42!

```
* (guess)
50
* (smaller)
25
* (bigger)
37
* (bigger)
43
* (smaller)
40
* (bigger)
41
* (bigger)
42
* (start-over)
50
*
```

Another Example

A bank account program:

Deposit:

- Update balance, return new balance.

Withdraw:

- Print “insufficient funds”, return balance unchanged if withdrawal exceeds balance
- Limit \$10000 per withdrawal
- No negative withdrawals
- Else, subtract withdrawal amount, return new balance

```
(defvar *balance* 100)

(defun deposit (amount)
  (setf *balance* (+ *balance* amount))
  *balance* )

(defun withdraw (amount)
  (if (< *balance* amount)
      (print "Insufficient funds")
      (if (> amount 10000)
          (print "Withdrawal exceeds limit")
          (if (< amount 0)
              (print "Withdrawal cannot be negative")
              (setf *balance* (- *balance* amount))
            )
        )
    )
  *balance*)
```


λ bank-account.lisp ×

λ bank-account.lisp > ...

```
1  (defvar *balance* 100)
2
3  (defun deposit (amount)
4    (setf *balance* (+ *balance* amount))
5    *balance* )
6
7  (defun withdraw (amount)
8    (if (< *balance* amount)
9        (print "Insufficient funds")
10       (if (> amount 10000)
11           (print "Withdrawal exceeds limit")
12           (if (< amount 0)
13               (print "Withdrawal cannot be negative")
14               (setf *balance* (- *balance* amount))
15           )
16       )
17    )
18    *balance*
19  )
20
21
```

```
* (deposit 200)
300
* (withdraw 50)
250
* (withdraw 5000000)
"Insufficient funds"
250
* (withdraw 500)
"Insufficient funds"
250
* (withdraw -500)
"Withdrawal cannot be negative"
250
* (deposit 10000)
10250
* (withdraw 10001)
"Withdrawal exceeds limit"
10250
* []
```

In Summary

- Course Intro
- Lisp intro, Lisp basics

Next class:

- More advanced Lisp
- Complexity analysis

