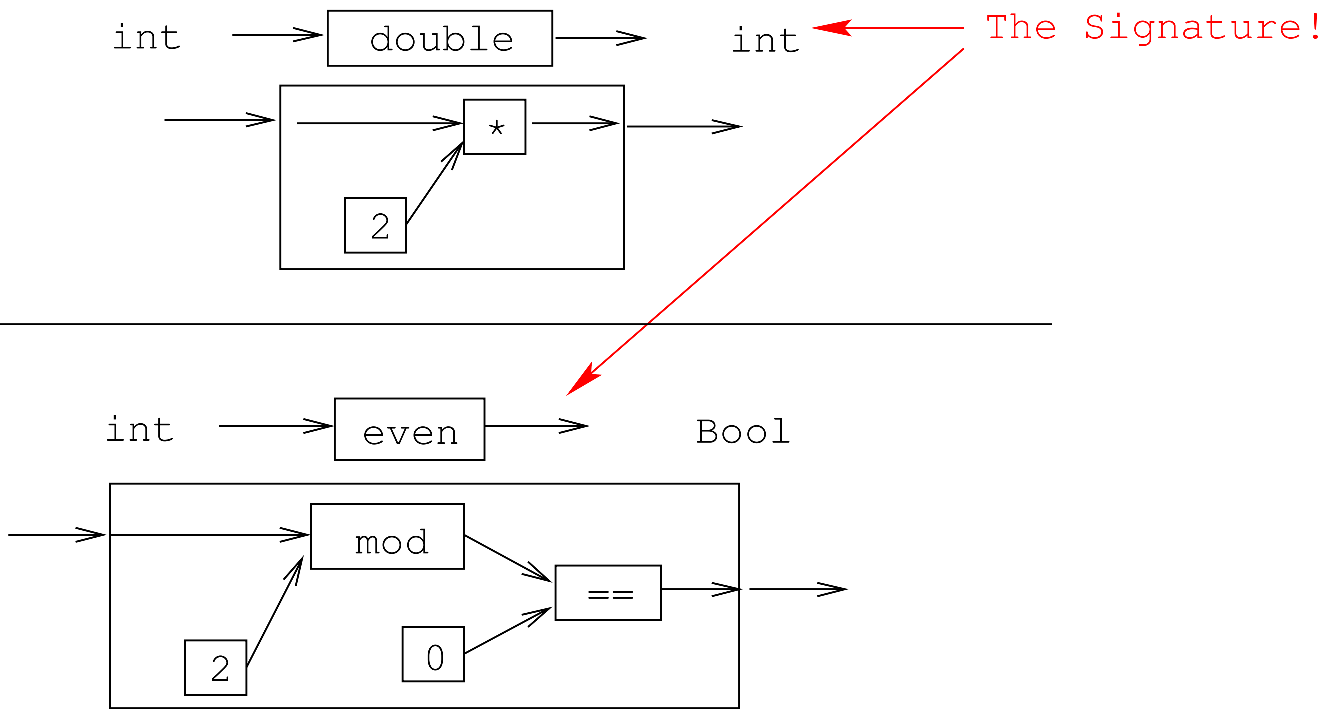
Function Signatures

To define a function we must specify the types of the input and output sets (domain and range, i.e. the function's signature), and an algorithm that maps inputs to outputs.



Referential Transparency

- The most important concept of functional programming is referential transparency. Consider the expression

$$(2 * 3) + 5 * (2 * 3)$$

- $(2 * 3)$ occurs twice in the expression, but it has the same meaning (6) both times.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- This concept occurs naturally in mathematics, but is broken by imperative programming languages.
- RT makes functional programs easier to reason about mathematically.

Referential Transparency. . .

- RT is a fancy term for a very simple concept.
- What makes FP particularly simple, direct, and expressive, is that there is only one mechanism (function application) for communicating between subprograms.
- In imperative programs we can communicate either through procedure arguments or updates of global variables. This is hard to reason about mathematically.
- A notation (programming language) where the value of an expression depends only on the values of the sub-expressions is called referentially transparent.

Referential Transparency…

- Pure functional programming languages are referentially transparent.
- This means that it is easy to find the meaning (value) of an expression.
- We can evaluate it by substitution. I.e. we can replace a function application by the function definition itself.

Referential Transparency...

Evaluate $\boxed{\texttt{even (double 5)}}$:

```
double x = 2 * x
even x = x mod 2 == 0
```

```
even (double 5) ⇒
          even (2 * 5) ⇒
          even 10 ⇒
          10 mod 2 == 0 ⇒
          0 == 0 ⇒ True
```

Referential Transparency...

So, isn't Pascal referentially transparent??? Well, sometimes, yes, but not always. If a Pascal function f has side-effects (updating a global variable, doing input or output), then $f(3) + f(3)$ may not be the same as $2 * f(3)$. I.e. The second $f(3)$ has a different meaning than the first one.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

```
begin
   G := 0;
   print f(3)+f(3);
   {prints 4+5=9}
end.
```

```
begin
   G := 0;
   print 2*f(3);
   {prints 2*4=8}
end.
```

Referential Transparency…

Furthermore, in many imperative languages the order in which the arguments to a binary operator are evaluated are undefined.

```
var G : integer;
function f (n:integer) :  integer;
begin G:=G+1; f:=G+n; end;
```

```
Left f(3) evaluated first.
begin
   G := 0;
   print f(3)-f(4);
   prints 4-6=-2
end.
```

```
Right f(3) evaluated first.
begin
   G := 0;
   print f(3)-f(4);
   prints 5-5=0
end.
```

Referential Transparency…

This cannot happen in a pure functional language.

1. Expressions and sub-expressions always have the same value, regardless of the environment in which they're evaluated.
2. The order in which sub-expressions are evaluated doesn't effect the final result.
3. Functions have no side-effects.
4. There are no global variables.

Referential Transparency…

5. Variables are similar to variables in mathematics: they hold a value, but they can't be updated.
6. Variables aren't (updatable) containers the way they are imperative languages.
7. Hence, functional languages are much more like mathematics than imperative languages. Functional programs can be treated as mathematical text, and manipulated using common algebraic laws.

Readings and References

- Read Scott, pp. 589–593.

Homework

- Here is a mathematical definition of the combinatorial function $\binom{n}{r}$ “n choose r”, which computes the number of ways to pick r objects from n:

$$\begin{pmatrix} n \\ r \end{pmatrix} = \frac{n!}{r! * (n-r)!}$$

- Give an extensional, intentional, and graphical definition of the combinatorial function, using the notations suggested in this lecture.
- You may want to start by defining an auxiliary function to compute the factorial function, $n! = 1 * 2 * \cdots * n$.

SCHEM E — INTRODUCTION

Background

- Scheme is based on LISP which was developed by John McCarthy in the mid 50s.
- LISP stands for *LISt Processing*, not *Lots of Irritating Silly Parentheses*.
- Functions and data share the same representation: S-Expressions.
- A basic LISP implementation needs six functions `cons, car, cdr, equal, atom, cond`.
- Scheme was developed by Sussman and Steele in 1975.

S-Expressions

- An S-Expression is a balanced list of parentheses.

More formally, an S-expression is

1. a literal (i.e., number, boolean, symbol, character, string, or empty list).
2. a list of s-expressions.

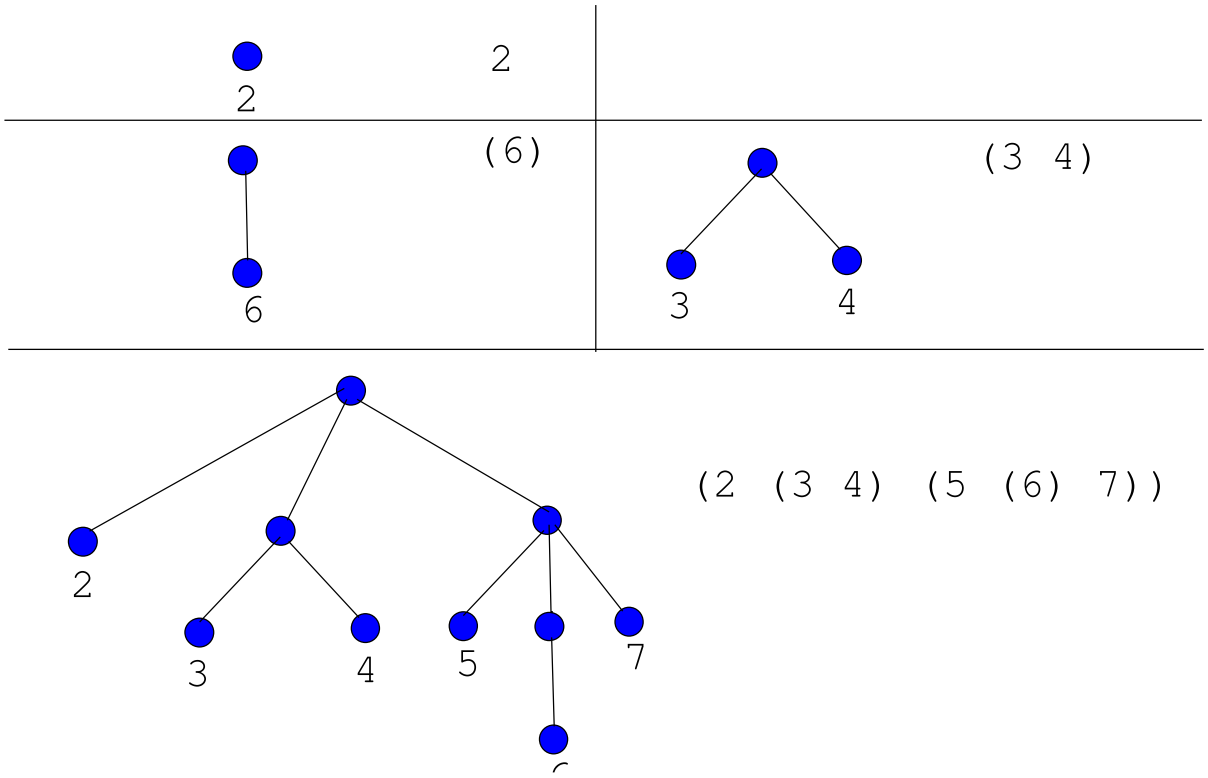
- Literals are sometimes called atoms.

S-Expressions — Examples

Legal	Illegal
`66`	`(`
`()`	`(5))`
`(4 5)`	`()()`
`((5))`	`(4 (5)`
`(()())`	`)(`
`((4 5) (6 (7)))`	

S-Expressions as Trees

- An S-expression can be seen as a linear representation of tree-structure:



S-Expressions as Function Calls

- A special case of an S-expression is when the first element of a list is a function name.
- Such an expression can be evaluated.

```
> (+ 4 5)
9
> (add-five-to-my-argument 20)
25
> (draw-a-circle 20 45)
#t
```

S-Expressions as Functions

- As we will see, function definitions are also S-expressions:

```
(define (farenheit-2-celsius f)
  (* (- f 32) 5/9)
)
```

- So, Scheme really only has one syntactic structure, the S-expression, and that is used as a data-structure (to represent lists, trees, etc), as function definitions, and as function calls.

Function Application

- In general, a function application is written like this:

$$(\texttt{operator}\ \texttt{arg}_1\ \texttt{arg}_2\ \ldots\ \texttt{arg}_n)$$

- The evaluation proceeds as follows:
  1. Evaluate `operator`. The result should be a function $\mathcal{F}$.
  2. Evaluate

     $$\texttt{arg}_1,\ \texttt{arg}_2,\ \ldots\ \texttt{arg}_n$$

     to get

     $$\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$$
  3. Apply $\mathcal{F}$ to $\texttt{val}_1,\ \texttt{val}_2,\ \ldots\ \texttt{val}_n$.

Function Application — Examples

```
> (+ 4 5)
9
> (+ (+ 5 6) 3)
14
> 7
7
> (4 5 6)
eval:  4 is not a function
> #t
#t
```

Atoms — Numbers

Scheme has

- Fractions (`5/9`)
- Integers (`5435`)
- Complex numbers (`5+2i`)
- Inexact reals (`#i3.14159265`)

```
> (+ 5 4)
9
> (+ (* 5 4) 3)
23
> (+ 5/9 4/6)
1.2
> 5/9
0.5
```

Atoms — Numbers. . .

```
> (+ 5/9 8/18)
1
> 5+2i
5+2i
> (+ 5+2i 3-i)
8+1i
> (* 236542164521634 3746573426573425643)
886222587860913289285513763860662
> pi
#i3.141592653589793
> e
#i2.718281828459045
> (* 2 pi)
#i6.283185307179586
```

Atoms — Numbers. . .

- Scheme tries to do arithmetic exactly, as much as possible.
- Any computations that depend on an inexact value becomes inexact.
- Scheme has many builtin mathematical functions:

```
> (sqrt 16)
4
> (sqrt 2)
#i1.4142135623730951
> (sin 45)
#i0.8509035245341184
> (sin (/ pi 2))
#i1.0
```

Atoms — Strings

- A string is enclosed in double quotes.

```
> (display "hello")
hello
> "hello"
"hello"
> (string-length "hello")
5
> (string-append "hello" " " "world!")
"hello world!"
```

Atoms — Booleans

- `true` is written `#t`.
- `false` is written `#f`.

```
> #t
true
> #f
false
> (display #t)
#t
> (not #t)
false
```

Identifiers

- Unlike languages like C and Java, Scheme allows identifiers to contain special characters, such as `! $ % & * + - . / : < = > ? @ ^ _ ~` . Identifiers should not begin with a character that can begin a number.
- This is a consequence of Scheme's simple syntax.
- You couldn't do this in Java because then there would be many ways to interpret the expression `X-5+Y`.

Legal	Illegal
`h-e-l-l-o`	`3some`
`give-me!`	`-stance`
`WTF?`	

Defining Variables

- define binds an expression to a global name:

  (define name *expression*)

```
(define PI 3.14)

> PI
3.14

(define High-School-PI (/ 22 7))

> High-School-PI
3.142857
```

Defining Functions

- define binds an expression to a global name:

  (define (name $\texttt{arg}_1$ $\texttt{arg}_2$ ...) *expression*)

- $\texttt{arg}_1$ $\texttt{arg}_2$ ... are formal function parameters.

```
(define (f) 'hello)

> (f)
hello

(define (square x) (* x x))

> (square 3)
9
```

Defining Helper Functions

- A Scheme program consists of a large number of functions.
- A function typically is defined by calling other functions, so called helper or auxiliary functions.

```
(define (square x) (* x x))

(define (cube x) (* x (square x)))

> (cube 3)
27
```

Preventing Evaluation

- Sometimes you don't want an expression to be evaluated.
- For example, you may want to think of (+ 4 5) as a list of three elements `+`, `4`, and `5`, rather than as the computed value `9`.
- (quote (+ 4 5)) prevents `(+ 4 5)` from being evaluated. You can also write '(+ 4 5).

```
> (display (+ 4 5))
9
> (display (quote (+ 4 5)))
(+ 4 5)
> (display '(+ 4 5))
(+ 4 5)
```

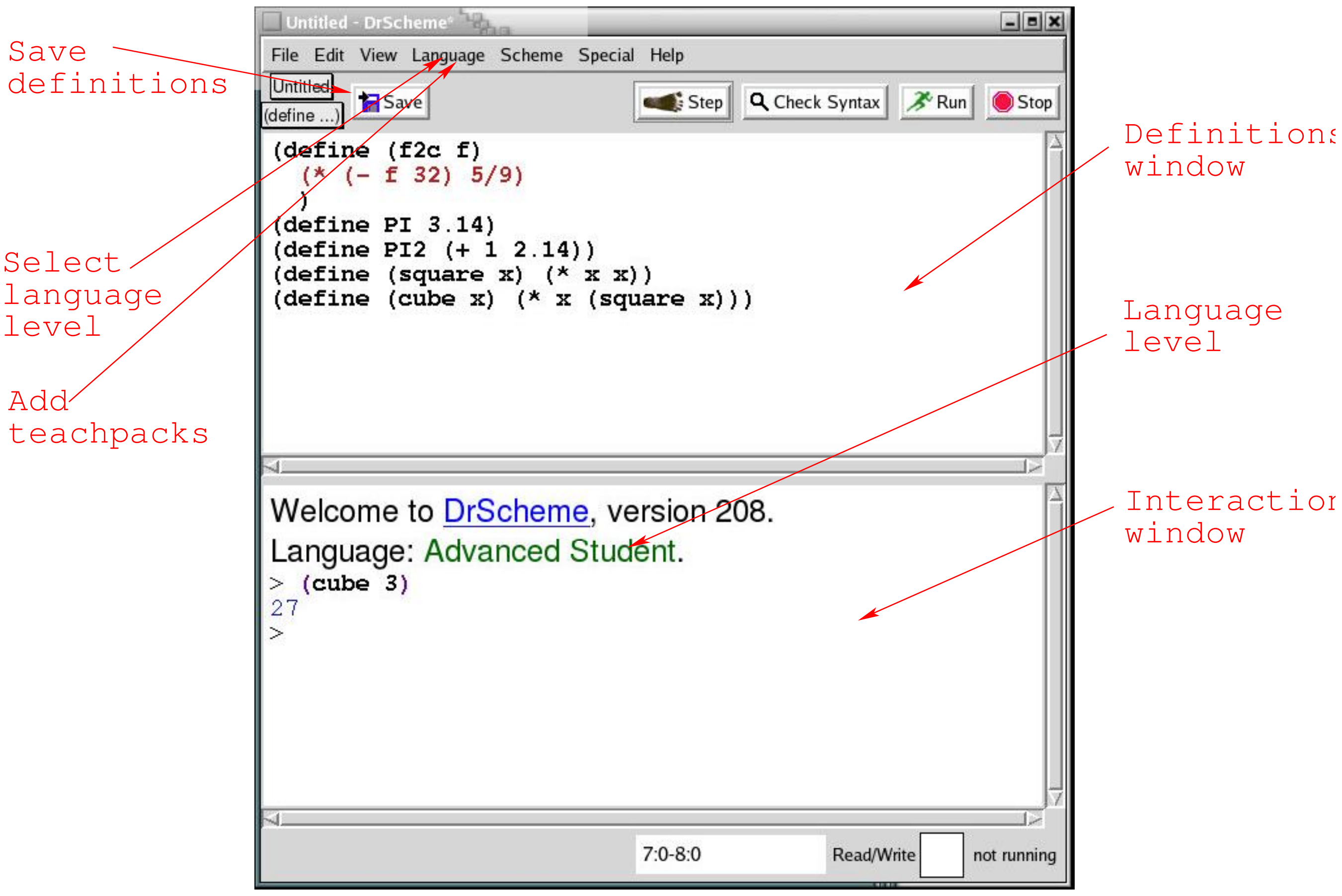
Dr Scheme

- Download DrScheme from here: `http://www.drscheme.org`.
- It has already been installed for you in lectura and the Windows machines in the lab.
- Start DrScheme under unix (on lectura) by saying

```
> drscheme
```

- On Windows and MacOS it may be enough to click on the DrScheme logo to start it up.

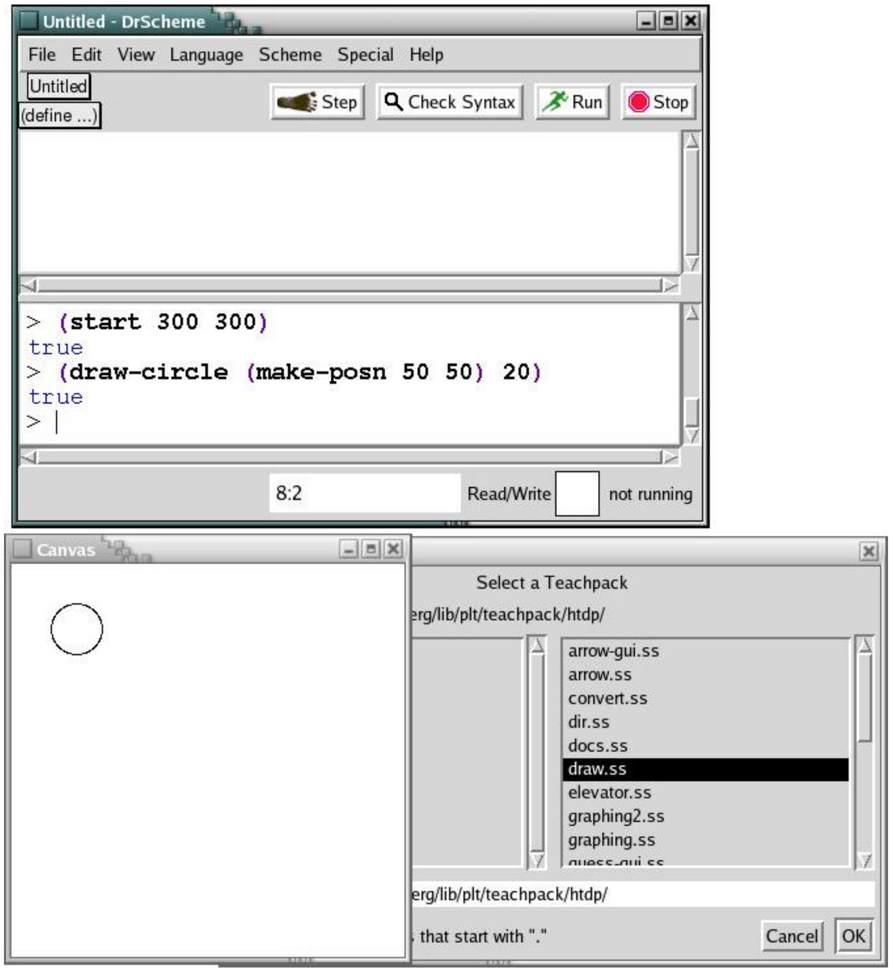
Dr Scheme



```
(define (f2c f)
  (* (- f 32) 5/9)
  )
(define PI 3.14)
(define PI2 (+ 1 2.14))
(define (square x) (* x x))
(define (cube x) (* x (square x)))
```

```
Welcome to DrScheme, version 208.
Language: Advanced Student.
> (cube 3)
27
>
```

Dr Scheme — Using TeachPacks



```
> (start 300 300)
true
> (draw-circle (make-posn 50 50) 20)
true
> |
```

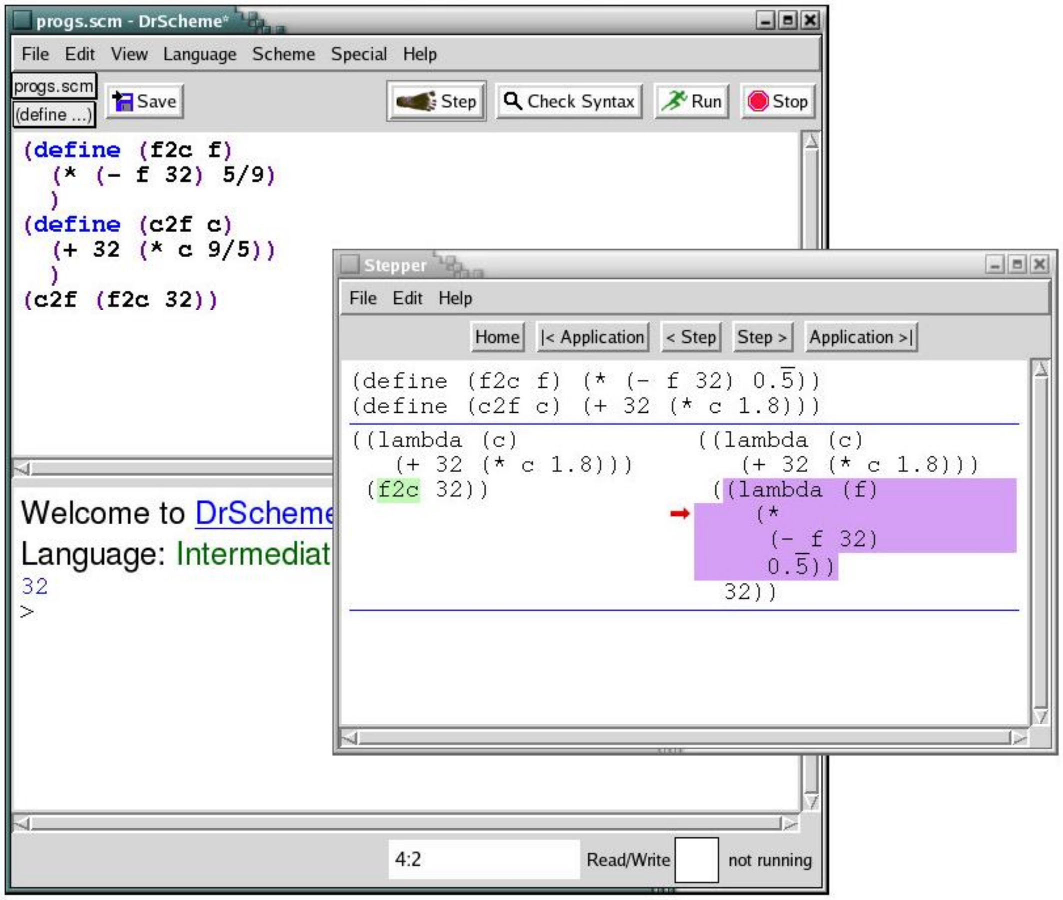
erg/lib/plt/teachpack/htdp/

```
arrow-gui.ss
arrow.ss
convert.ss
dir.ss
docs.ss
draw.ss
elevator.ss
graphing2.ss
graphing.ss
guess-gui.ss
```

erg/lib/plt/teachpack/htdp/

that start with "."

Dr Scheme — Using the Stepper



```
(define (f2c f)
  (* (- f 32) 5/9)
  )
(define (c2f c)
  (+ 32 (* c 9/5))
  )
(c2f (f2c 32))
```

```
Welcome to DrScheme
Language: Intermediat
32
>
```

```
(define (f2c f) (* (- f 32) 0.5))
(define (c2f c) (+ 32 (* c 1.8)))
```

```
((lambda (c)                ((lambda (c)
   (+ 32 (* c 1.8)))           (+ 32 (* c 1.8)))
 (f2c 32))                   ((lambda (f)
                                (*
                                 (- f 32)
                                 0.5))
                              32))
```

References

- Read Scott, pp. 523-527, 528-539.
- Free interpreter: http://www.drscheme.org.
- Manual:

  http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme.html
- Tutorials:
  - http://ai.uwaterloo.ca/~dale/cs486/s99/scheme-tutorial.html
  - http://cs.wwc.edu/%7Ecs_dept/KU/PR/Scheme.html
  - http://www.cis.upenn.edu/%7Eungar/CIS520/scheme-tutorial.html
- http://dmoz.org/Computers/Programming/Languages/Lisp/Scheme

References. . .

- Language reference manual: http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps.
- Some of this material is taken from http://www.ecf.utoronto.ca/~gower/CSC326F/slides, ©Diana Inkpen 2002, Suzanne Stevenson 2001.

Scheme so Far

- A function is defined by

```
    (define (name arguments) expression)
```

- A variable is defined by

```
            (define name expression)
```

- Strings are inclosed in double quotes, like `"this"`. Common operations on strings are
  - (string-length `string`)
  - (string-append `list-of-strings`)
- Numbers can be exact integers, inexact reals, fractions, and complex. Integers can get arbitrarily large.
- Booleans are written `#t` and `#f`.

Scheme so Far…

- An inexact number is written: `#i3.14159265`.
- Common operations on numbers are
  - (+ `arg1 arg2`), (- `arg1 arg2`)
  - (add1 `arg`), (sub1 `arg`)
  - (min `arg1 arg2`), (max `arg1 arg2`)
- A function application is written:

```
> (function-name arguments)
```

- Quoting is used to prevent evaluation

```
(quote argument)
```

or

```
'argument
```

SCHEM E — HISTORY

History of the Lisp Language

http://www.apl.jhu.edu/~hall/text/Papers/Brief-History-of-Lisp.ps

The following information is derived from the history section of dpANS Common Lisp.

Lisp is a family of languages with a long history. Early key ideas in Lisp were developed by John McCarthy during the 1956 Dartmouth Summer Research Project on Artificial Intelligence. McCarthy's motivation was to develop analgebraic list processing language for artificial intelligence work. Implementation efforts for early dialects of Lisp were undertaken on the IBM 704, the IBM 7090, the Digital Equipment Corporation (DEC) PDP-1, the DECPDP-6, and the PDP-10. The primary dialect of Lisp between 1960 and 1965 was Lisp 1.5. By the early 1970's there were two predominant dialects of Lisp, both arising from these early efforts: MacLisp and Interlisp. Forfurther information about very early Lisp dialects, see The Anatomy of Lisp or Lisp 1.5 Programmer's Manual.

History of the Lisp Language…

MacLisp improved on the Lisp 1.5 notion of special variables and error handling. MacLisp also introduced theconcept of functions that could take a variable number of arguments, macros, arrays, non-local dynamic exits, fast arithmetic, the first good Lisp compiler, and an emphasis on execution speed.

Interlisp introduced many ideas into Lisp programming environments and methodology. One of the Interlisp ideasthat influenced Common Lisp was an iteration construct implemented by Warren Teitelman that inspired the loop macro used both on the Lisp Machines and in MacLisp, and now in Common Lisp.

History of the Lisp Language…

Although the first implementations of Lisp were on the IBM 704 and the IBM 7090, later work focussed on theDEC PDP-6 and, later, PDP-10 computers, the latter being the mainstay of Lisp and artificial intelligence work at such places as Massachusetts Institute of Technology (MIT), Stanford University, and Carnegie Mellon University(CMU) from the mid-1960's through much of the 1970's. The PDP-10 computer and its predecessor the PDP-6 computer were, by design, especially well-suited to Lisp because they had 36-bit words and 18-bit addresses. Thisarchitecture allowed a cons cell to be stored in one word; single instructions could extract the car and cdr parts. The PDP-6 and PDP-10 had fast, powerful stack instructions that enabled fast function calling. But the limitations ofthe PDP-10 were evident by 1973: it supported a small number of researchers using Lisp, and the small, 18-bit address space (262,144 36-bit words) limited the size of a single program. One response to the address spaceproblem was the Lisp Machine, a special-purpose computer designed to run Lisp programs. The other response was to use general-purpose computers with address spaces larger than 18 bits, such as the DEC VAX and the S-1 MarkIIA.

History of the Lisp Language…

The Lisp machine concept was developed in the late 1960’s. In the early 1970’s, Peter Deutsch, working withDaniel Bobrow, implemented a Lisp on the Alto, a single-user minicomputer, using microcode to interpret a byte-code implementation language. Shortly thereafter, Richard Greenblatt began work on a different hardwareand instruction set design at MIT. Although the Alto was not a total success as a Lisp machine, a dialect of Interlisp known as Interlisp-D became available on the D-series machines manufactured by Xerox—the Dorado,Dandelion, Dandetiger, and Dove (or Daybreak). An upward-compatible extension of MacLisp called Lisp Machine Lisp became available on the early MIT Lisp Machines. Commercial Lisp machines from Xerox, LispMachines (LMI), and Symbolics were on the market by 1981. During the late 1970’s, Lisp Machine Lisp began to expand towards a much fuller language. Sophisticated lambdalists, setf, multiple values, and structures like those in Common Lisp are the results of early experimentation with programming styles by the Lisp Machine group.

History of the Lisp Language. . .

JonI White and others migrated these features to MacLisp. Around 1980, Scott Fahlman and others at CMU began work on a Lisp to run on the Scientific Personal Integrated Computing Environment (SPICE) workstation. One of the goals of the project was to design a simpler dialect thanLisp Machine Lisp.

The Macsyma group at MIT began a project during the late 1970's called the New Implementation of Lisp (NIL)for the VAX, which was headed by White. One of the stated goals of the NIL project was to fix many of the historic, but annoying, problems with Lisp while retaining significant compatibility with MacLisp.

History of the Lisp Language. . .

Richard P. Gabriel began the design of a Lisp to run on the S-1 Mark IIA supercomputer. S-1 Lisp, nevercompletely functional, was the test bed for adapting advanced compiler techniques to Lisp implementation. Eventually the S-1 and NIL groups collaborated.

The first effort towards Lisp standardization was made in 1969, when Anthony Hearn and Martin Griss at theUniversity of Utah defined Standard Lisp—a subset of Lisp 1.5 and other dialects—to transport REDUCE, a symbolic algebra system. During the 1970's, the Utah group implemented first a retargetable optimizing compilerfor Standard Lisp, and then an extended implementation known as Portable Standard Lisp (PSL). By the mid 1980's, PSL ran on about a dozen kinds of computers.

History of the Lisp Language. . .

PSL and Franz Lisp—a MacLisp-like dialect for Unix machines—were the first examples of widely availableLisp dialects on multiple hardware platforms. One of the most important developments in Lisp occurred during the second half of the 1970's: Scheme. Scheme,designed by Gerald J. Sussman and Guy L. Steele Jr., is a simple dialect of Lisp whose design brought to Lisp some of the ideas from programming language semantics developed in the 1960's. Sussman was one of the primeinnovators behind many other advances in Lisp technology from the late 1960's through the 1970's. The major contributions of Scheme were lexical scoping, lexical closures, first-class continuations, and simplified syntax (noseparation of value cells and function cells). Some of these contributions made a large impact on the design of Common Lisp. For further information about Scheme, see IEEE Standard for the Scheme Programming Languageor "Revised^4 Report on the Algorithmic Language Scheme."

History of the Lisp Language…

In the late 1970's object-oriented programming concepts started to make a strong impact on Lisp. At MIT, certainideas from Smalltalk made their way into several widely used programming systems. Flavors, an object-oriented programming system with multiple inheritance, was developed at MIT for the Lisp machine community byHoward Cannon and others. At Xerox, the experience with Smalltalk and Knowledge Representation Language (KRL) led to the development of Lisp Object Oriented Programming System (LOOPS) and later Common LOOPS. These systems influenced the design of the Common Lisp Object System (CLOS). CLOS was developedspecifically for X3J13's standardization effort, and was separately written up in "Common Lisp Object System Specification." However, minor details of its design have changed slightly since that publication, and that papershould not be taken as an authoritative reference to the semantics of the Common Lisp Object System.

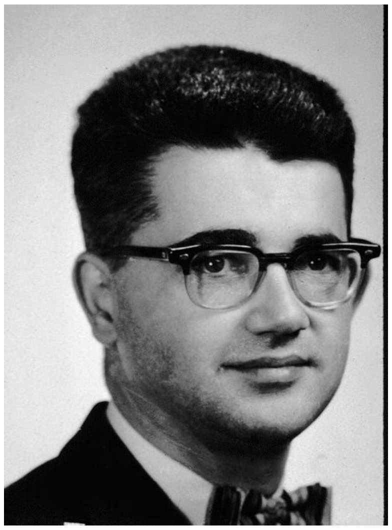
History of the Lisp Language…

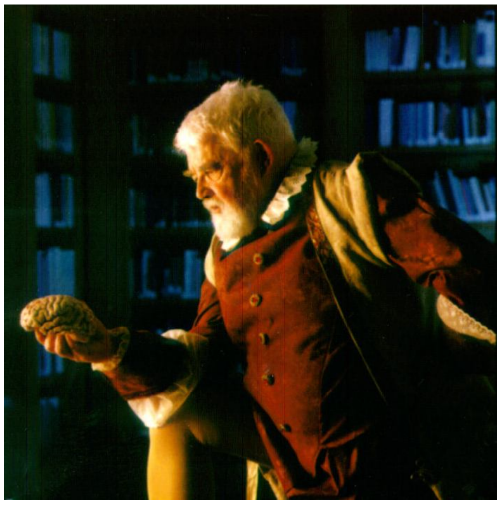
In 1980 Symbolics and LMI were developing Lisp Machine Lisp; stock-hardware implementation groups weredeveloping NIL, Franz Lisp, and PSL; Xerox was developing Interlisp; and the SPICE project at CMU was developing a MacLisp-like dialect of Lisp called SpiceLisp. In April 1981, after a DARPA-sponsored meeting concerning the splintered Lisp community, Symbolics, theSPICE project, the NIL project, and the S-1 Lisp project joined together to define Common Lisp. Initially spearheaded by White and Gabriel, the driving force behind this grassroots effort was provided by Fahlman, DanielWeinreb, David Moon, Steele, and Gabriel. Common Lisp was designed as a description of a family of languages. The primary influences on Common Lisp were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, andScheme. Common Lisp: The Language is a description of that design. Its semantics were intentionally underspecified in places where it was felt that a tight specification would overly constrain Common Lisp researchand use.

History of the Lisp Language…

In 1986 X3J13 was formed as a technical working group to produce a draft for an ANSI Common Lisp standard.Because of the acceptance of Common Lisp, the goals of this group differed from those of the original designers. These new goals included stricter standardization for portability, an object-oriented programming system, acondition system, iteration facilities, and a way to handle large character sets. To accommodate those goals, a new language specification was developed.

John McCarthy





John McCarthy has been Professor of Computer Science at Stanford University since 1962. His research is mainly in artificial intelligence. Long ago he originated the Lisp programming language and the initial research on general purpose time-sharing computer systems.

http://www-formal.stanford.edu/jmc/personal.html

Guy Steele

Guy L. Steele Jr. is a Distinguished Engineer at Sun Microsystems, Inc. He received his A.B. in applied mathematics from Harvard College (1975), and his S.M. and Ph.D. in computer science and artificial intelligence from M.I.T. (1977 and 1980). He has also been an assistant professor of computer science at Carnegie-Mellon University; a member of technical staff at Tartan Laboratories in Pittsburgh, Pennsylvania; and a senior scientist at Thinking Machines Corporation. He joined Sun Microsystems in 1994.

Guy Steele...

The Association for Computing Machinery awarded him the 1988 Grace Murray Hopper Award and named him an ACM Fellow in 1994.

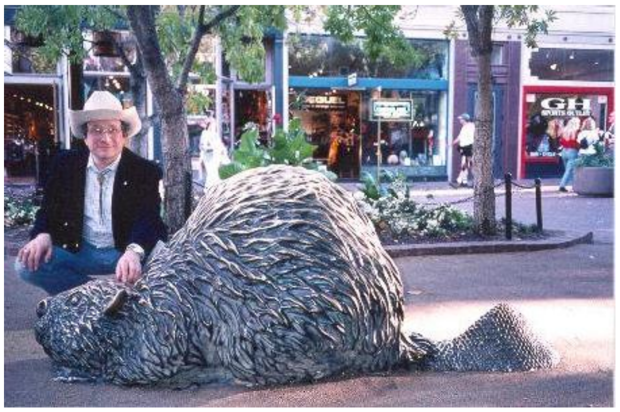
He has served on accredited standards committees X3J11 (C language) and X3J3 (Fortran) and is currently chairman of X3J13 (Common Lisp). He was also a member of the IEEE committee that produced the IEEE Standard for the Scheme Programming Language, IEEE Std 1178-1990.

He has had chess problems published in Chess Life and Review and is a Life Member of the United States Chess Federation. He has sung in the bass section of the MIT Choral Society and the Masterworks Chorale as well as in choruses with the Pittsburgh Symphony Orchestra at Great Woods and with the Boston Concert Opera. He has played the role of Lun Tha in The King and I and the title role in Li'l Abner. He designed the original EMACS command set and was the first person to port TeX.

http://www.sls.csail.mit.edu/˜hurley/guysteele.html

http://encyclopedia.thefreedictionary.com/Guy%20Steele

Gerald Jay Sussman



Gerald Jay Sussman is the Matsushita Professor of Electrical Engineering at the Massachusetts Institute of Technology. He received the S.B. and the Ph.D. degrees in mathematics from the Massachusetts Institute of Technology in 1968 and 1973, respectively. He has been involved in artificial intelligence research at M.I.T. since 1964. His research has centered on understanding the problem-solving strategies used by scientists and engineers, with the goals of automating parts of the process and formalizing it to provide more effective methods of science and engineering education. Sussman has also worked in computer languages, in computer architecture and in VLSI design.

Gerald Jay Sussman…

Sussman is a coauthor (with Hal Abelson and Julie Sussman) of the introductory computer science textbook used at M.I.T. The textbook, "Structure and Interpretation of Computer Programs," has been translated into French, German, Chinese, Polish, and Japanese. As a result of this and other contributions to computer-science education, Sussman received the ACM's Karl Karlstrom Outstanding Educator Award in 1990, and the Amar G. Bose award for teaching in 1991.

Sussman's contributions to Artificial Intelligence include problem solving by debugging almost-right plans, propagation of constraints applied to electrical circuit analysis and synthesis, dependency-based explanation and dependency-based backtracking, and various language structures for expressing problem-solving strategies. Sussman and his former student, Guy L. Steele Jr., invented the Scheme programming language in 1975.

`http://www.swiss.ai.mit.edu/~gjs/gjs.html`

Beating the Averages

Paul Graham
(This article is based on a talk given at the Franz Developer Symposium in Cambridge, MA, on March 25, 2001.)

http://paulgraham.com/avg.html

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface. $[\cdots]$ Another unusual thing about this software was that it was written primarily in a programming language called Lisp.

Beating the Averages

It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. Lisp gave us a great advantage over competitors using less powerful languages. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. $[\cdots]$ Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. $[\cdots]$

Beating the Averages

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. $[\cdots]$ What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have.

Beating the Averages

But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.
$[\cdots]$by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. $[\cdots]$
I'll begin with a shockingly controversial statement: programming languages vary in power.

Beating the Averages

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers. $[\cdots]$

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to.

Beating the Averages

[· · ·] The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening– that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried. [· · ·]

Exercise

```
sed 's/Lisp/Scheme/g'
```

SCHEME — CONDITIONAL EXPRESSIONS

Comparison Functions

- Boolean functions (by convention) end with a ?.
- We can discriminate between different kinds of numbers:

```
> (complex?  3+4i)
   #t
> (complex?  3)
   #t
> (real?  3)
   #t
> (real?  -2.5+0.0i)
   #t
> (rational?  6/10)
```

Comparison Functions…

```
    #t
> (rational?  6/3)
    #t
> (integer?  3+0i)
    #t
> (integer?  3.0)
    #t
> (integer?  8/4)
    #t
```

Tests on Numbers

- Several of the comparison functions can take multiple arguments.
- `(< 4 5 6 7 9 234)` returns true since the numbers are monotonically increasing.

```
> (< 4 5)
true
> (< 4 5 6 7 9 234)
true
> (> 5 2 1 3)
false
> (= 1 1 1 1 1)
true
> (<= 1 2 2 2 3)
true
```

Tests on Numbers…

```
> (>= 5 5)
true
> (zero?  5)
false
> (positive?  5)
true
> (negative?  5)
false
> (odd?  5)
true
> (even?  5)
false
```

Conditionals — If

- If the `test-expression` evaluates to `#f` (False) return the valuen of the `then-expression`, otherwise return the value of the `else-expression`:

```
(if test-expression
   then-expression
   else-expression
)
```

- Up to language level “Advanced Student” if-expressions must have two parts.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior, where the else-expression is optional.

Conditionals — If…

```
> (define x 5)
> (if (= x 5) 2 4)
2
> (if (< x 3)
        (display "hello")
        (display "bye"))
bye
> (display
        (if (< x 3) "hello" "bye"))
bye
```

If it's not False (`#f`), it's True (`#t`)

- Any value that is not false, is interpreted as true.
- NOTE: In DrScheme this depends on which language level you set. Up to "Advanced Student", the `test-expression` of an `if` must be either `#t` or `#f`.
- Set the language level to Standard (R5RS) to get the standard Scheme behavior:

```
> (if 5 "hello" "bye")
"hello"
> (if #f "hello" "bye")
"bye"
> (if #f "hello")
> (if #t "hello")
"hello"
```

Boolean Operators

- `and` and `or` can take multiple arguments.
- `and` returns true if none of its arguments evaluate to False.
- `or` returns true if any of its arguments evaluates to True.

```
> (and (< 3 5) (odd?  5) (inexact?  (cos 32)))
#t
> (or (even?  5) (zero?  (- 5 5)))
#t
> (not 5)
#f
> (not #t)
#f
```

Boolean Operators…

- In general, any value that is not `#f` is considered true.
- `and` and `or` evaluate their arguments from left to right, and stop as soon as they know the final result.
- The last value evaluated is the one returned.

```
> (and "hello")
"hello"
> (and "hello" "world")
"world"
> (or "hello" "world")
"hello"
```

Defining Boolean Functions

- We can define our own boolean functions:

```
(define (big-number?  n)
      (> n 10000000)
)

> (big-number?  5)
#f
> (big-number?  384783274832748327)
#t >
```

Conditionals — `cond`

- `cond` is a generalization of `if`:

  (cond
      (*cond-expression*$_1$ *result-expression*$_1$)
      (*cond-expression*$_2$ *result-expression*$_2$)
      ...
      (else *else-expression*))

- Each `cond-expression`$_i$ is evaluated in turn, until one evaluates to not False.

```
> (cond
          ((< 2 3) 4)
          ((= 2 3) 5)
          (else 6))
4
```

Conditionals — `cond`...

- To make this a bit more readable, we use square brackets around the `cond`-clauses:

```
(cond
   [cond-expr_1 result-expr_1]
   [cond-expr_2 result-expr_2]
   ...
   [else else-expression])

> (cond [#f 5] [#t 6])
6
> (cond
    [(= 4 5) "hello"]
    [(> 4 5) "goodbye"]
    [(< 4 5) "see ya!"])
"see va!"
```

Conditionals — `case`

- `case` is like Java/C's `switch` statment:

```
(case key
   [(expr1 expr2 ...)   result-expr1]
   [(expr11 expr11 ...)   result-expr2]
   ...
   (else else-expr))
```

- The *key* is evaluated once, and compared against each *cond-expr* in turn, and the corresponding *result-expr* is returned.

```
> (case 5 [(2 3) "hello"] [(4 5) "bye"])
"bye"
```

Conditionals — case...

```
(define (classify n)
    (case n
        [(2 4 8 16 32) "small power of 2"]
        [(2 3 5 7 11) "small prime number"]
        [else "some other number"]
    )
 )
> (classify 4)
"small power of 2"
> (classify 3)
"small prime number"
> (classify 2)
"small power of 2"
> (classify 32476)
"some other number"
```

Sequencing

- To do more than one thing in sequence, use `begin`:

  (begin *arg*$_1$ *arg*$_2$ ...)

```
> (begin
    (display "the meaning of life=")
    (display (* 6 7))
    (newline)
)
the meaning of life=42
```

Examples — $!n$

- Write the factorial function $!n$:

```
(define (! n)
    (cond
        [(zero? n) 1]
        [else (* n (! (- n 1)))]
    )
)

> (! 5)
120
```

Examples — $\binom{n}{r}$

- Write the $\binom{n}{r}$ function in Scheme:

$$\binom{n}{r} = \frac{n!}{r! * (n-r)!}$$

- Use the factorial function from the last slide.

```
(define (choose n r)
   (/ (! n) (* (! r) (! (- n r))))
)

> (choose 5 2)
10
```

Examples — `(sum m n)`

- Write a function `(sum m n)` that returns the sum of the integers between `m` and `n`, inclusive.

```
(define (sum m n)
    (cond
        [(= m n) m]
        [else (+ m (sum (+ 1 m) n))]
    )
)

> (sum 1 2)
3
> (sum 1 4)
10
```

Examples — Ackermann's function

- Implement Ackermann's function:

$$\begin{aligned} A(1,j) &= 2j \text{ for } j \geq 1 \\ A(i,1) &= A(i-1,2) \text{ for } i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)) \text{ for } i,j \geq 2 \end{aligned}$$

```
(define (A i j)
   (cond
      [(and (= i 1) (>= j 1)) (* 2 j)]
      [(and (>= i 2) (= j 1)) (A (- i 1) 2)]
      [(and (>= i 2) (>= j 2))
            (A (- i 1) (A i (- j 1)))]
   )
)
```

Examples — Ackermann's function...

- Ackermann's function grows very quickly:

```
> (A 1 1)
2
> (A 3 2)
512
> (A 3 3)
1561585988519419914804999641169225
4958731641184786755447122887443528
0601470939536037485963338068553800
6371637297210170750776562389313989
2867298012168192
```

Scheme so Far

- Unlike languages like Java and C which are statically typed (we describe in the program text what type each variable is) Scheme is dynamically typed. We can test at runtime what particular type of number an atom is:
  - `(complex? arg), (real? arg)`
  - `(rational? arg), (integer? arg)`
- Tests on numbers:
  - `(< arg1, arg2), (> arg1, arg2)`
  - `(= arg1, arg2), (<= arg1, arg2)`
  - `(>= arg1, arg2), (zero? arg)`
  - `(positive? arg), (negative? arg)`
  - `(odd? arg), (even? arg)`

Scheme so Far…

- Unlike many other languages like Java which are statement-oriented, Scheme is expression-oriented. That is, every construct (even `if`, `cond`, etc) return a value. The `if-expression` returns the value of the `then-expr` or the `else-expr`:

  ```
  (if test-expr then-expr else-expr)
  ```

  depending on the value of the `test-expr`.

Scheme so Far…

- The `cond`-expression evaluates its guards until one evaluates to non-false. The corresponding value is returned:

```
(cond
    (guard₁ value₁)
    (guard₂ value₂)
    ...
    (else else-expr))
```

Scheme so Far. . .

- The `case`-expression evaluates `key`, finds the first matching expression, and returns the corresponding result:

```
(case key
   [(expr1  expr2  ...)   result-expr1]
   [(expr11  expr11  ...)   result-expr2]
   ...
   (else else-expr))
```

Scheme so Far…

- `and` and `or` take multiple arguments, evaluate their results left-to-right until the outcome can be determined (for `or` when the first non-`false`, for `and` when the first `false` is found), and returns the last value evaluated.

SCHEME — SYMBOLS AND STRUCTURES

Symbols

- In addition to numbers, strings, and booleans, Scheme has a primitive data-type (*atom*) called symbol.
- A symbol is a lot like a string. It is written:

```
'identifier
```

- Here are some examples:

```
'apple
'pear
'automobile
```

- `(symbol? arg)` checks if an atom is a symbol.
- To compare two symbols for equality, use `(eq? arg1 arg2)`. `HTDP` says to use `(symbol=? arg1 arg2)` but DrScheme doesn't seem to support this.

Symbols. . .

```
> (symbol?  "hello")
#f
> (symbol?  'apple)
#t
> (eq?  'a 'a)
#t
> (eq?  'a 'b)
#f
> (display 'apple)
apple
> (string->symbol "apple")
apple
> (symbol->string 'apple)
"apple"
```

Symbols. . .

```
(define (healthy?  f)
    (case f
        [(sushi sashimi) 'hell-yeah]
        [(coke) 'I-wish]
        [(licorice) 'no-but-yummy]
        [else 'nope]
    ))
> (healthy?  'sashimi)
hell-yeah
> (healthy?  'coke)
i-wish
> (healthy?  'licorice)
no-but-yummy
> (healthy?  'pepsi)
nope
```

Structures

- Some versions of Scheme have structures. Select Advanced Student in DrScheme.
- These are similar to C's `struct`, and Java's `class` (but without inheritance and methods).
- Use `define-struct` to define a structure:

```
(define-struct struct-name (f1 f2 ...))
```

- `define-struct` will automatically define a constructor:

```
(make-struct-name (f1 f2 ...))
```

  and field-selectors:

```
struct-name-f1
struct-name-f2
```

Structures...

```
(define-struct person (name sex date-of-birth))

> (define bob (make-person "bob" 'male '1978))
> bob
(make-person "bob" 'male '1978)
> (define alice (
          make-person "alice" 'female '1979))

> (person-sex bob)
'male
> (person-date-of-birth alice)
'1979
```

Equivalence…

- `(equal?  a b)` returns #t if `a` and `b` are strings that print the same.
- This is known as structural equivalence.

```
> (equal?  "hello" "hello")
true
> (equal?  alice bob)
false
> (define alice1 (
          make-person "alice" 'female '1979))
> (define alice2 (
          make-person "alice" 'female '1979))
> (equal?  alice1 alice2)
true
```

SCHEM E — LIST PROCESSING

Constructing Lists

- The most important data structure in Scheme is the list.
- Lists are constructed using the function `cons`:

  ```
  (cons first rest)
  ```

  `cons` returns a list where the first element is `first`, followed by the elements from the list `rest`.

  ```
  > (cons 'a '())
  (a)
  > (cons 'a (cons 'b '()))
  (a b)
  > (cons 'a (cons 'b (cons 'c '())))
  (a b c)
  ```

Constructing Lists. . .

- There are a variety of short-hands for constructing lists.
- Lists are heterogeneous, they can contain elements of different types, including other lists.

```
> '(a b c)
(a b c)
> (list 'a 'b 'c)
(a b c)

> '(1 a "hello")
(1 a "hello")
```

Examining Lists

- `(car L)` returns the first element of a list. Some implementations also define this as `(first L)`.
- `(cdr L)` returns the list `L`, without the first element. Some implementations also define this as `(rest L)`.
- Note that `car` and `cdr` do not destroy the list, just return its parts.

```
> (car '(a b c))
'a
> (cdr '(a b c))
'(b c)
```

Examining Lists…

- Note that `(cdr L)` always returns a list.

```
> (car (cdr '(a b c)))
'b
> (cdr '(a b c))
'(b c)
> (cdr (cdr '(a b c)))
'(c)
> (cdr (cdr (cdr '(a b c))))
'()
> (cdr (cdr (cdr (cdr '(a b c)))))
error
```

Examining Lists. . .

- A shorthand has been developed for looking deep into a list:

  ```
  (clist of "a" and "d"r L)
  ```

  Each `"a"` stands for a `car`, each `"d"` for a `cdr`.

- For example, `(caddar L)` stands for

  ```
  (car (cdr (cdr (car L))))
  ```

  ```
  > (cadr '(a b c))
  'b
  > (cddr '(a b c))
  '(c)
  > (caddr '(a b c))
  'c
  ```

Lists of Lists

- Any S-expression is a valid list in Scheme.
- That is, lists can contain lists, which can contain lists, which...

```
> '(a (b c))
(a (b c))
> '(1 "hello" ("bye" 1/4 (apple)))
(1 "hello" ("bye" 1/4 (apple)))
> (caaddr '(1 "hello" ("bye" 1/4 (apple))))
"bye"
```

List Equivalence

- `(equal?  L1 L2)` does a structural comparison of two lists, returning `#t` if they “look the same”.
- `(eqv?  L1 L2)` does a “pointer comparison”, returning `#t` if two lists are “the same object”.

```
> (eqv?  '(a b c) '(a b c))
false
> (equal?  '(a b c) '(a b c))
true
```

List Equivalence. . .

- This is sometimes referred to as deep equivalence vs. shallow equivalence.

```
> (define myList '(a b c))
> (eqv?  myList myList)
true
> (eqv?  '(a (b c (d)) '(a (b c (d))))
false
> (equal?  '(a (b c (d)) '(a (b c (d))))
true
```

Predicates on Lists

- `(null? L)` returns `#t` for an empty list.
- `(list? L)` returns `#t` if the argument is a list.

```
> (null? '())
#t
> (null? '(a b c))
#f
> (list? '(a b c))
#t
> (list? "(a b c)")
#f
```

List Functions — Examples…

```
> (memq 'z '(x y z w))
#t
> (car (cdr (car '((a) b (c d)))))
(c d)
> (caddr '((a) b (c d)))
(c d)
> (cons 'a '())
(a)
> (cons 'd '(e))
(d e)
> (cons '(a b) '(c d))
((a b) (c d))
```

Recursion over Lists — cdr-recursion

- Compute the length of a list.
- This is called cdr-recursion.

```
(define (length x)
   (cond
      [(null?  x) 0]
      [else (+ 1 (length (cdr x)))]
   )
)

> (length ’(1 2 3))
3
> (length ’(a (b c) (d e f)))
3
```

Recursion over Lists — car-cdr-recursion

- Count the number of atoms in an S-expression.
- This is called car-cdr-recursion.

```
(define (atomcount x)
   (cond
      [(null?  x) 0]
      [(list?  x)
            (+ (atomcount (car x))
               (atomcount (cdr x)))]
      [else 1]
   ))
> (atomcount '(1))
1
> (atomcount '("hello" a b (c 1 (d))))
6
```

Recursion Over Lists — Returning a List

- Map a list of numbers to a new list of their absolute values.
- In the previous examples we returned an atom — here we're mapping a list to a new list.

```
(define (abs-list L)
    (cond
        [(null?  L) '()]
        [else (cons (abs (car L))
                        (abs-list (cdr L)))]
    )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```

Recursion Over Two Lists

- `(atom-list-eq?  L1 L2)` returns `#t` if `L1` and `L2` are the same list of atoms.

```
(define (atom-list-eq?  L1 L2)
   (cond
      [(and (null?  L1) (null?  L2)) #t]
      [(or (null?  L1) (null?  L2)) #f]
      [else (and
         (atom?  (car L1))
         (atom?  (car L2))
         (eqv?  (car L1) (car L2))
         (atom-list-eq?  (cdr L1) (cdr L2)))]
   )
)
```

Recursion Over Two Lists…

```
> (atom-list-eq? '(1 2 3) '(1 2 3))
#t
> (atom-list-eq? '(1 2 3) '(1 2 a))
#f
```

Append

```
(define (append L1 L2)
    (cond
        [(null?  L1) L2]
        [else
            (cons (car L1)
                (append (cdr L1) L2))]
    )
)

> (append '(1 2) '(3 4))
(1 2 3 4)
> (append '() '(3 4))
(3 4)
> (append '(1 2) '())
(1 2)
```

Deep Recursion — `equal?`

```
(define (equal? x y)
   (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x))
         (not (atom? y))
         (equal? (car x) (car y))
         (equal? (cdr x) (cdr y)))))
```

```
> (equal? 'a 'a)
#t
> (equal? '(a) '(a))
#t
> (equal? '((a)) '((a)))
#t
```

Patterns of Recursion — cdr-recursion

- We process the elements of the list one at a time.
- Nested lists are not descended into.

```
(define (fun L)
   (cond
      [(null?  L) return-value]
      [else ...(car L) ...(fun (cdr L)) ...]
   )
)
```

Patterns of Recursion — car-cdr-recursion

- We descend into nested lists, processing every atom.

```
(define (fun x)
   (cond
      [(null?  x) return-value]
      [(atom?  x) return-value]
      [(list?  x)
            ...(fun (car x)) ...
            ...(fun (cdr x)) ...]
      [else return-value]
   ))
```

Patterns of Recursion — Maps

- Here we map one list to another.

```
(define (map L)
   (cond
      [(null?  L) '()]
      [else (cons (...(car L) ...)
                  (map (cdr L)))]
   )
)
```

Example: Binary Trees

- A binary tree can be represented as nested lists:

```
(4 (2 () ()) (6 (5 () ()) ()))
```

- Each node is represented by a triple

  `(`*`data left-subtree right-subtree`*`)`

- Empty subtrees are represented by `()`.

Example: Binary Trees. . .

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))

(define (print-spaces N)
   (cond
      [(= N 0) ""]
      [else (begin
         (display " ")
         (print-spaces (- N 1)))]))

(define (print-tree tree)
   (print-tree-rec tree 0))
```

Example: Binary Trees. . .

```
(define (print-tree-rec tree D)
   (cond
      [(null?  tree)]
      [else (begin
         (print-spaces D)
         (display (key tree)) (newline)
         (print-tree-rec (left tree) (+ D 1))
         (print-tree-rec (right tree) (+ D 1))
)]))

> (print-tree '(4 (2 () ()) (6 (5 () ()) ())))
4
   2
   6
      5
```

Binary Trees using Structures

- We can use structures to define tree nodes.

```
(define-struct node (data left right))

(define (tree-member x T)
    (cond
        [(null?  T) #f]
        [(= x (node-data T)) #t]
        [(< x (node-data T))
            (tree-member x (node-left T))]
        [else
            (tree-member x (node-right T))]
    )
)
```

Binary Trees using Structures...

```
(define tree
    (make-node 4
        (make-node 2 '() '())
        (make-node 6
            (make-node 5 '() '())
            (make-node 9 '() '())))))

> (tree-member 4 tree)
true
> (tree-member 5 tree)
true
> (tree-member 19 tree)
false
```

Homework

- Write a function `swapFirstTwo` which swaps the first two elements of a list. Example: `(1 2 3 4)` $\Rightarrow$ `(2 1 3 4)`.
- Write a function `swapTwoInLists` which, given a list of lists, forms a new list of all elements in all lists, with first two of each swapped. Example: `((1 2 3) (4) (5 6))` $\Rightarrow$ `(2 1 3 4 6 5)`.