7054 Programmiersprachen

Prof. O. Nierstrasz

Wintersemester 1997/98

Table of Contents

12. Scripting *223*

13. Summary, Trends, Research ... *241*

1. Programming Languages

Lecturer: Prof. O. Nierstrasz
Schützenmattstr. 14/103; Tel. 631.4618; oscar@iam.unibe.ch
Secr.: Frau I. Huber, Tel. 631.4692
Assistants: F. Achermann, D. Kuehni
WWW: http://iamwww.unibe.ch/~scg/Lectures/

Text:

- Kenneth C. Louden, *Programming Languages: Principles and Practice*, PWS Publishing (Boston), 1993.

Other Sources:

- *PostScript® Language Tutorial and Cookbook*, Adobe Systems Incorporated, Addison-Wesley, 1985
- Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, pp 359-411.
- Clocksin and Mellish, *Programming in Prolog*, Springer Verlag, 1981.
- Guido van Rossum, *Python Reference Manual*, Stichting Mathematisch Centrum, Amsterdam, 1996.

Schedule

10.23	1. Introduction	
10.30	2. Stack-based Programming	— *Postscript*
11.06	3. Functional Programming	— *Haskell*
11.13	4. Type systems	
11.20	5. An application of Functional Programming	
11.27	6. Lambda Calculus	
12.04	7. Fixed Points; Other Calculi	
12.11	8. Programming language semantics	
12.18	9. Logic Programming	— *Prolog*
01.08	10. Applications of Logic Programming	
01.15	11. Symbolic Interpretation	
01.22	12. Scripting Languages	— *Python ...*
01.29	13. Summary, Trends, Research ...	
02.05	*Final exam*	

Themes Addressed in this Course

Paradigms:

- What computational paradigms are supported by modern, high-level programming languages?
- How well do these paradigms match classes of programming problems?

Abstraction

- How do different languages abstract away from the low-level details of the underlying hardware implementation?
- How do different languages support the specification of software abstractions needed for a specific task?

Types

- How do type systems help in the construction of flexible, reliable software?

Semantics

- How can one formalize the meaning of a programming language?
- How can semantics aid in the implementation of a programming language?

What is a Programming Language?

- ☞ A formal language for describing *computation*
- ☞ A “user interface” to a computer
- ☞ “Turing tar pit” — equivalent computational power
- ☞ Programming paradigms — different expressive power
- ☞ Syntax + semantics
- ☞ Compiler, or interpreter, or translator

How do Programming Languages Differ?

Generations (increasing abstraction; imperative → declarative):

- ❑ 1GL: machine codes
- ❑ 2GL: symbolic assemblers
- ❑ 3GL: (machine independent) imperative languages (FORTRAN, Pascal ...)
- ❑ 4GL: domain specific application generators

Common Constructs:

- ☞ basic data types (numbers, etc.); variables; expressions; statements; keywords; control constructs; procedures; comments; errors ...

Uncommon Constructs:

- ☞ type declarations; special types (strings, arrays, matrices, ...); sequential execution; concurrency constructs; packages/modules; objects; general functions; generics; modifiable state; ...

Programming Paradigms

A programming language is a *problem-solving tool.*

Imperative style:

- ☞ program = algorithms + data

Functional style:

- ☞ program = functions ○ functions

Logic programming style:

- ☞ program = facts + rules

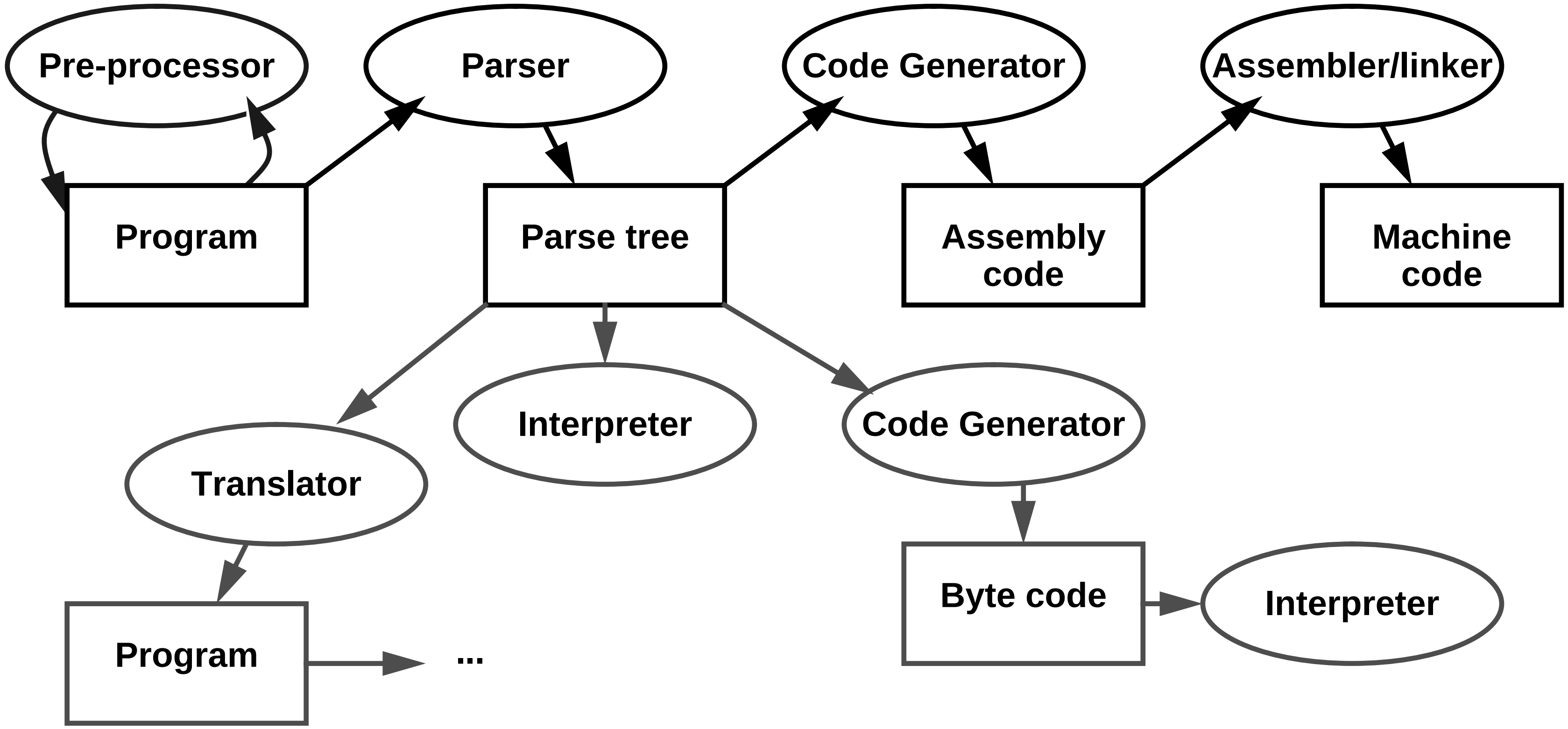
Object-oriented style:

- ☞ program = objects + messages

Other styles and paradigms: blackboard, pipes and filters, constraints, lists, ...

Compilers and Interpreters

Compilers and interpreters have similar front-ends, but have different back-ends:



Details will differ, but the general scheme remains the same ...

A Brief Chronology

Early 1950s "order codes" (primitive assemblers)

1957	FORTRAN	*the first high-level programming language*
1958	ALGOL	*the first modern, imperative language*
1960	LISP, COBOL	
1962	APL, SIMULA	*the birth of OOP (SIMULA)*
1964	BASIC, PL/I	
1966	ISWIM	*first modern functional language (proposal)*
1970	Prolog	*logic programming is born*
1972	C	*the systems programming language*
1975	Pascal, Scheme	
1978	CSP	
1978	FP	
1980	dBASE II	
1983	Smalltalk-80, Ada	*OOP is reinvented*
1984	Standard ML	*FP becomes mainstream (?)*
1986	C++, Eiffel	*OOP is reinvented (again)*
1988	CLOS, Mathematica, Oberon	
1990	Haskell	*FP is reinvented*
1995	Java	*OOP is reinvented for the internet*

Fortran

History:

- John Backus (1953) sought to write programs in conventional mathematical notation, and generate code comparable to good assembly programs
  - ☞ No language design effort (made it up as they went along)
  - ☞ Most effort spent on code generation and optimization
  - ☞ FORTRAN I released April 1957; working by April 1958
  - ☞ Current standards are FORTRAN 77 and FORTRAN 90

Innovations:

- comments
- assignments to variables of complex expressions
- **DO** loops
- Symbolic notation for subroutines and functions
- Input/output formats
- machine-independence

Successes:

- Easy to learn; high level
- Promoted by IBM; addressed large user base (scientific computing)

ALGOL 60

History:

- ❑ Committee of PL experts formed in 1955 to design universal, machine-independent, algorithmic language
- ❑ First version (ALGOL 58) never implemented; criticisms led to ALGOL 60

Innovations:

- ❑ BNF (Backus-Naur Form) introduced to define syntax (led to syntax-directed compilers)
- ❑ First block-structured language; variables with local scope
- ❑ Variable size arrays
- ❑ Structured control statements
- ❑ Recursive procedures

Successes:

- ❑ Never displaced FORTRAN, but highly influenced design of other PLs

COBOL

History:

- designed by committee of US computer manufacturers
- targeted business applications
- intended to be readable by managers

Innovations:

- separate descriptions of environment, data, and processes

Successes:

- Adopted as *de facto* standard by US DOD
- Stable standard for 25 years
- Still the most widely used PL for business applications

4GLs

“Problem-oriented” languages

- PLs for “non-programmers”
- Very High Level (VHL) languages for specific problem domains

Classes of 4GLs (no clear boundaries):

- Report Program Generator (RPG)
- Application generators
- Query languages
- Decision-support languages

Successes:

- highly popular, but generally *ad hoc*

PL/I

History:

- ❑ designed by committee of IBM and users (early 1960s)
- ❑ intended as (large) general-purpose language for broad classes of applications

Innovations:

- ❑ Support for concurrency (but not synchronization)
- ❑ exception-handling by **on** conditions

Successes:

- ❑ achieved both run-time efficiency and flexibility (at expense of complexity)
- ❑ first “complete” general purpose language

Interactive Languages

Made possible by advent of time-sharing systems (early 1960s through mid 1970s).

BASIC:

- ❑ developed at Dartmouth College in mid 1960s
- ❑ minimal; easy to learn
- ❑ incorporated basic O/S commands (NEW, LIST, DELETE, RUN, SAVE)

APL:

- ❑ developed by Ken Iverson for *concise* description of numerical algorithms
- ❑ large, non-standard alphabet (52 characters in addition to alphanumerics)
- ❑ primitive objects are *arrays* (lists, tables or matrices)
- ❑ operator-driven (power comes from composing array operators)
- ❑ no operator precedence (statements parsed right to left)

Special-Purpose Languages

SNOBOL:

- ❑ first successful string manipulation language
- ❑ influenced design of text editors more than other PLs
- ❑ string operations: pattern-matching and substitution
- ❑ arrays and associative arrays (tables)
- ❑ variable-length strings

Lisp:

- ❑ performs computations on symbolic expressions
- ❑ symbolic expressions are represented as lists
- ❑ small set of constructor/selector operations to create and manipulate lists
- ❑ recursive rather than iterative control
- ❑ no distinction between data and programs
- ❑ first PL to implement storage management by garbage collection
- ❑ affinity with lambda calculus

Functional Languages

ISWIM (If you See What I Mean):

- Peter Landin (1968) — paper proposal

FP:

- John Backus (1978) — Turing award lecture

ML:

- Edinburgh
- initially designed as meta-language for theorem proving
- Hindley-Milner type inference
- “non-pure” functional language (with assignments/side effects)

Miranda, Haskell:

- “pure” functional languages with “lazy evaluation”

Prolog

History:

- ❑ originated at U. Marseilles (early 1970s), and compilers developed at Marseilles and Edinburgh (mid to late 1970s)

Innovations:

- ❑ theorem proving paradigm
- ❑ programs as sets of clauses: facts, rules and questions
- ❑ computation by “unification”

Successes:

- ❑ prototypical logic programming language
- ❑ used in Japanese Fifth Generation Initiative

Object-Oriented Languages

History:

- Simula was developed by Nygaard and Dahl (early 1960s) in Oslo as a language for simulation programming, by adding *classes* and *inheritance* to ALGOL 60
- Smalltalk was developed by Xerox PARC (early 1970s) to drive graphic workstations

Innovations:

- encapsulation of data and operations (contrast ADTs)
- inheritance to share behaviour and interfaces

Successes:

- Smalltalk project pioneered OO user interfaces ...
- Large commercial impact since mid 1980s
- Countless new languages: C++, Objective C, Eiffel, Beta, Oberon, Self, Perl 5, Python, Java, Ada 95 ...

Scripting Languages

History:

- ❑ Countless “shell languages” and “command languages” for operating systems and configurable applications
- ❑ Unix shell (ca. 1971) developed as user shell and scripting tool
- ❑ HyperTalk (1987) was developed at Apples to script HyperCard stacks
- ❑ TCL (1990) developed as embedding language and scripting language for X windows applications (via Tk)

Innovations:

- ❑ Pipes and filters (Unix shell)
- ❑ Generalized embedding/command languages (TCL)

Successes:

- ❑ Unix Shell, awk, emacs, HyperTalk, AppleTalk, TCL, Python, Perl, ...

Summary

You should know the answers to these questions:

- ☐ What, exactly, is a programming language?
- ☐ How do compilers and interpreters differ?
- ☐ Why was FORTRAN developed?
- ☐ What were the main achievements of ALGOL 60?
- ☐ Why do we call Pascal a "Third Generation Language"?
- ☐ What is a "Fourth Generation Language"?

Can you answer the following questions?

- ✎ *Why are there so many programming languages?*
- ✎ *Why are FORTRAN and COBOL still important programming languages?*
- ✎ *What language would you use to implement a spelling checker? A filter to translate upper-to-lower case? A theorem prover? An address database? An expert system? A game server for initiating chess games on the internet? A user interface for a network chess client?*

2. Stack-based Programming

Overview

- PostScript objects, types and stacks
- Arithmetic operators
- Graphics operators
- Procedures and variables
- Arrays and dictionaries

References:

- *PostScript® Language Tutorial and Cookbook*, Adobe Systems Incorporated, Addison-Wesley, 1985
- *PostScript® Language Reference Manual*, Adobe Systems Incorporated, second edition, Addison-Wesley, 1990

PostScript

PostScript “is a simple interpretive programming language ... to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages.”

- ❑ introduced in 1985 by Adobe
- ❑ display standard now supported by all major printer vendors
- ❑ simple, stack-based programming language
- ❑ minimal syntax
- ❑ large set of built-in operators
- ❑ PostScript programs are usually generated from applications, rather than hand-coded
- ❑ three language variants:
  - *Level 1:* the original 1985 PostScript
  - *Level 2:* additional support for dictionaries, memory management ...
  - *Display PostScript:* special support for screen display

Syntax

- Comments: from “%” to next newline or formfeed

  `% This is a comment`

- Numbers: signed integers, reals and radix numbers

  `123 -98 0 +17 -.002 34.5 123.6e10 1E-5 8#1777 16#FFE 2#1000`

- Strings: text in parentheses or hexadecimal in angle brackets

  `(Special characters are escaped: \n \t \( \) \\ ...)`

- Names: tokens that consist of “regular characters” but aren’t numbers

  `abc Offset $$ 23A 13-456 a.b $MyDict @pattern`

- Literal names: start with slash

  `/buffer /proc`

- Arrays: enclosed in square brackets

  `[ 123 /abc (hello) ]`

- Procedures: enclosed in curly brackets

  `{ add 2 div } % add top two stack elements and divide by 2`

Semantics

The PostScript interpreter manages four stacks representing the execution state of a PostScript program:

- *Operand* stack:
  - holds (arbitrary) operands and results of PostScript operators
- *Dictionary* stack:
  - holds only dictionaries where keys and values may be stored
- *Execution* stack:
  - holds executable objects (e.g. procedures) in stages of execution
- *Graphics* state stack:
  - keeps track of current coordinates etc.

A PostScript program is a sequence of tokens, representing typed objects, that is interpreted to manipulate the four stacks and the display.

Object types

Every object is either *literal* or *executable*:

- *Literal objects* are pushed on the operand stack:
  - integers, reals, string constants, literal names, arrays, procedures

- *Executable objects* are interpreted:
  - built-in operators
  - names bound to procedures (in the current dictionary context)

Simple Object Types: are copied by *value*

- boolean, fontID, integer, name, null, operator, real ...

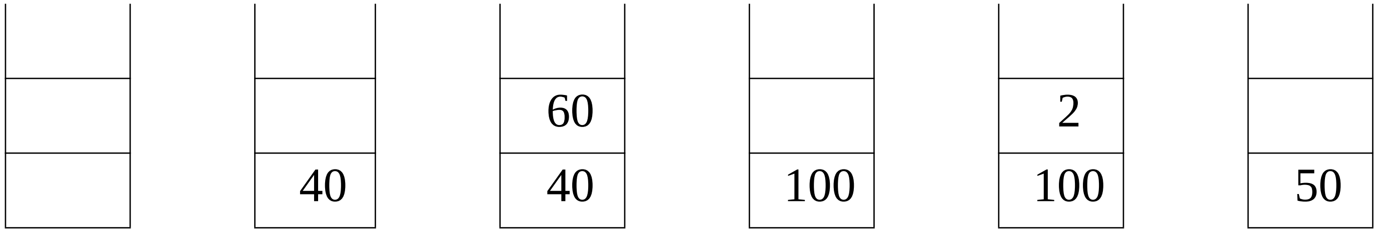
Composite Object Types: are copied by *reference*

- array, dictionary, string ...

The operand stack

Compute the average of 40 and 60:

```
40 60 add 2 div
```



At the end, the result is left on the top of the operand stack.

Stack and arithmetic operators

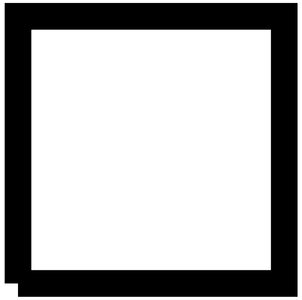
num_1 num_2	**add**	sum	$num_1 + num_2$
num_1 num_2	**sub**	difference	$num_1 - num_2$
num_1 num_2	**mul**	product	$num_1 * num_2$
num_1 num_2	**div**	quotient	num_1 / num_2
int_1 int_2	**idiv**	quotient	integer divide
int_1 int_2	**mod**	remainder	int_1 mod int_2
num den	**atan**	angle	arctangent of *num/den*
any	**pop**	-	discard top element
any_1 any_2	**exch**	any_2 any_1	exchange top two elements
any	**dup**	any any	duplicate top element
any_1 ... any_n n	**copy**	any_1 ... any_n any_1 ... any_n	duplicate top n elements
any_n ... any_0 n	**index**	any_n ... any_0 any_n	duplicate $n+1$th element

Other arithmetic operators: abs, neg, ceiling, floor, round, truncate, sqrt, cos, sin, exp, ln, log, rand, srand, rrand

Drawing a Box

“A *path* is a set of straight lines and curves that define a region to be filled or a trajectory that is to be drawn on the *current page*.”

```
newpath             % clear the current drawing path
100 100 moveto      % move to (x,y) coordinate (100,100)
100 200 lineto      % draw a line to coordinate (100,200)
200 200 lineto
200 100 lineto
100 100 lineto
10 setlinewidth     % set the width for drawing lines
stroke              % draw along the current path
showpage            % and display the current page
```



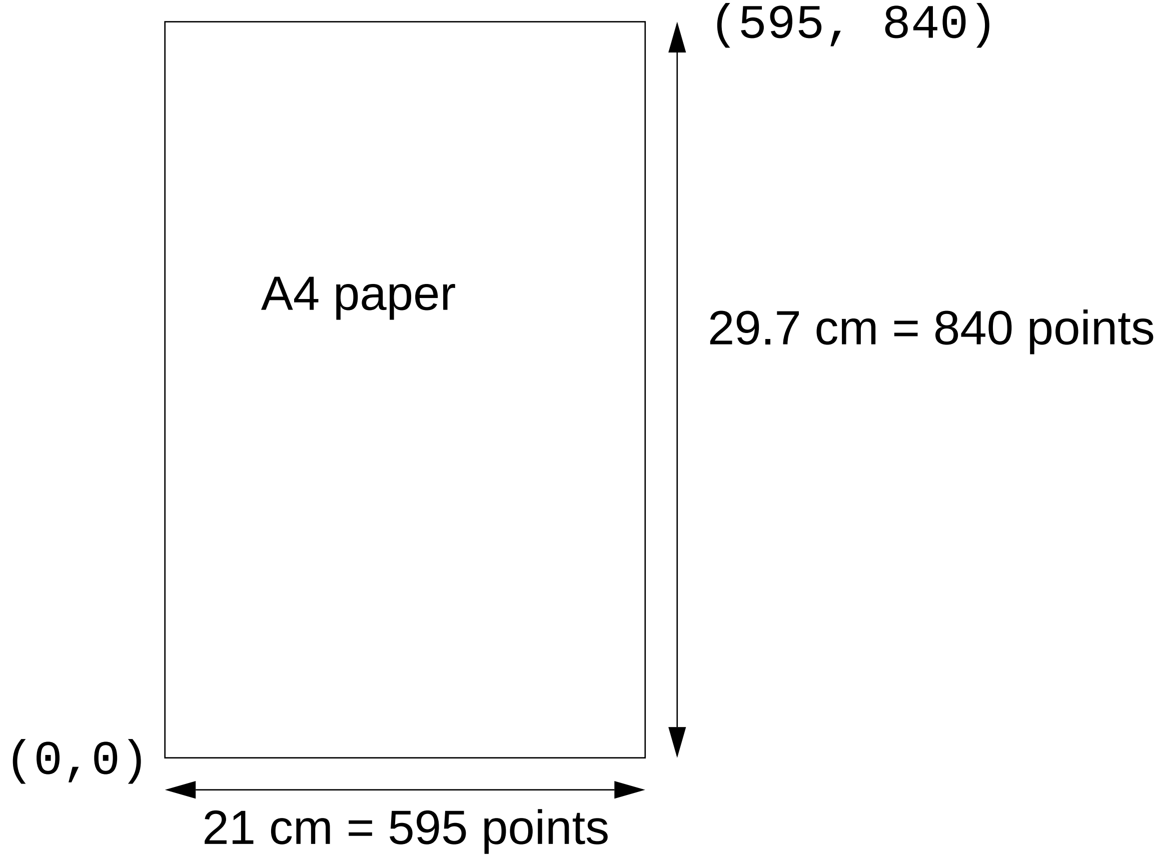
Path construction operators

-	**newpath**	-	initialize current path to be empty
-	**currentpoint**	x y	return current coordinates
x y	**moveto**	-	set current point to *(x, y)*
dx dy	**rmoveto**	-	relative moveto
x y	**lineto**	-	append straight line to *(x, y)*
dx dy	**rlineto**	-	relative lineto
x y r ang_1 ang_2	**arc**	-	append counterclockwise arc
-	**closepath**	-	connect subpath back to start
-	**fill**	-	fill current path with current colour
-	**stroke**	-	draw line along current path
-	**showpage**	-	output and reset current page

Coordinates

Coordinates are measured in *points:*

☞ 72 points = 1 inch = 2.54 cm.



(595, 840)

A4 paper

29.7 cm = 840 points

(0,0)

21 cm = 595 points

Hello World

Before you can print text, you must (1) look up the desired font, (2) scale it to the required size, and (3) set it to be the *current font*.

```
/Times-Roman findfont                % look up the Times Roman font
   18 scalefont                      % scale it to 18 points
   setfont                           % set this to be the current font
100 500 moveto                       % go to coordinate (100, 500)
(Hello world) show                   % draw the string “Hello world”
showpage                             % render the current page
```

```
Hello world
```

Character and font operators

key	**findfont**	font	return font dict identified by *key*
font scale	**scalefont**	font’	scale *font* by *scale* to produce *font’*
font	**setfont**	-	set font dictionary
-	**currentfont**	font	return current font
string	**show**	-	print *string*
string	**stringwidth**	w_x w_y	width of *string* in current font

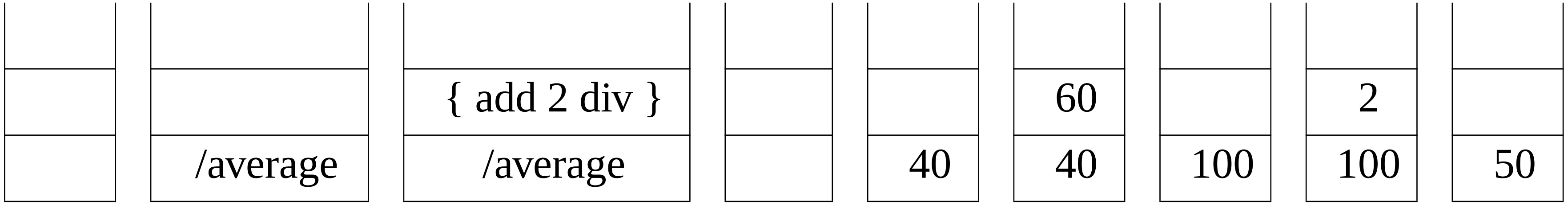
Procedures and Variables

Variables and procedures are defined by binding names to literal or executable objects.

key value	**def**	-	associate *key* and *value* in current dictionary

Define a general procedure to compute averages:

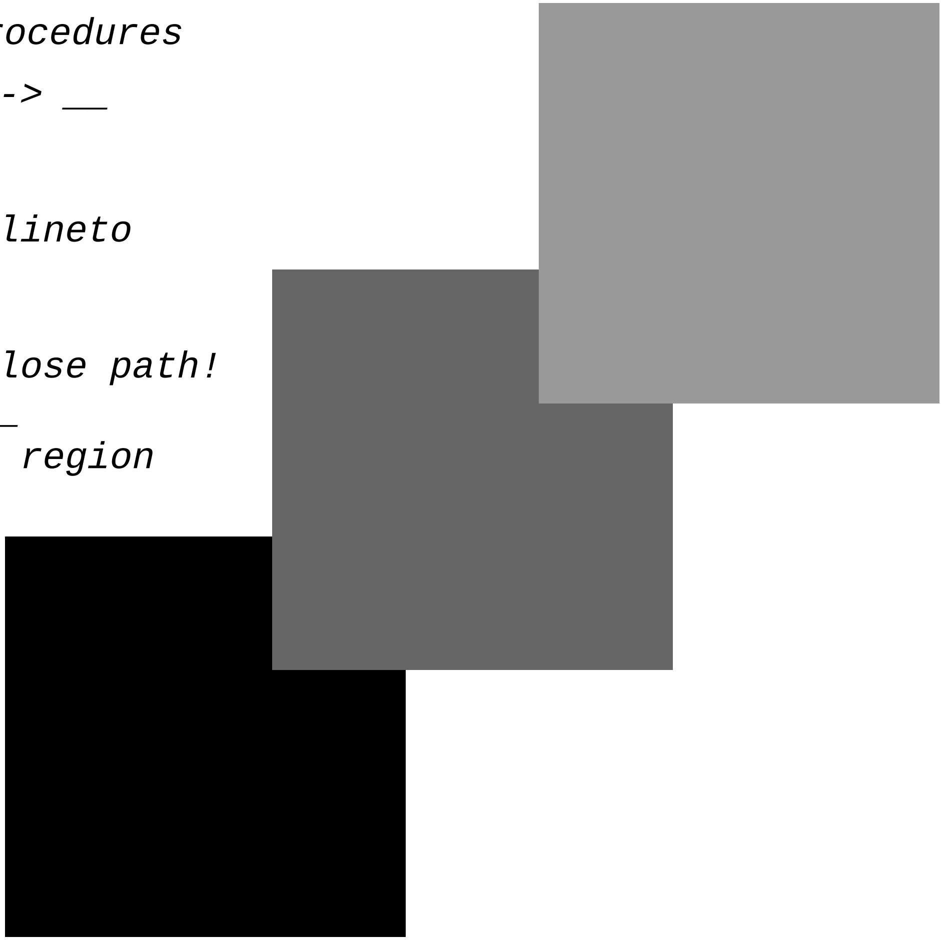
```
/average { add 2 div } def      % bind the name “average” to “{ add 2 div }”
40 60 average
```



A Box procedure

Most PostScript programs are separated into a *prologue* and a *script*.

```
% Prologue -- application specific procedures
/box {                       % grey x y -> __
   newpath
   moveto                    % x y -> __
   0 150 rlineto             % relative lineto
   150 0 rlineto
   0 -150 rlineto
   closepath                 % cleanly close path!
   setgray                   % grey -> __
   fill                      % colour in region
} def
% Script -- usually generated
0 100 100 box
0.4 200 200 box
0.6 300 300 box
0 setgray
showpage
```

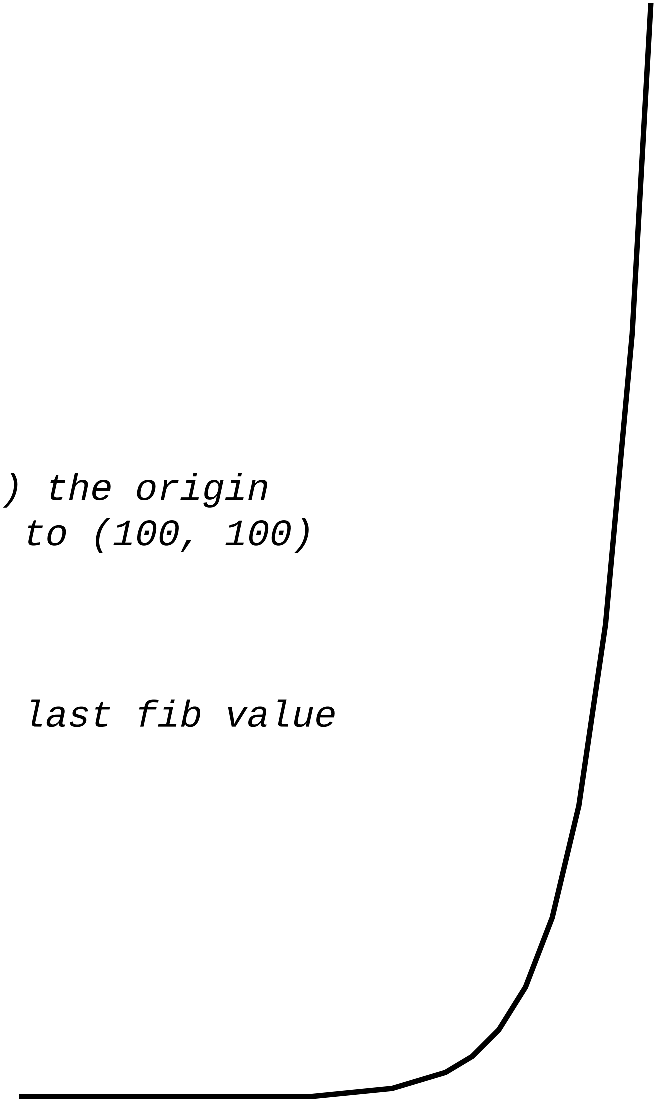


Graphics state and coordinate operators

num	**setlinewidth**	-	set line width
num	**setgray**	-	set colour to gray value from 0 (black) to 1 (white)
s_x s_y	**scale**	-	scale use space by s_x and s_y
angle	**rotate**	-	rotate user space by *angle* degrees
t_x t_y	**translate**	-	translate user space by (t_x, t_y)
-	**matrix**	matrix	create identity matrix
matrix	**currentmatrix**	matrix	fill *matrix* with CTM
matrix	**setmatrix**	-	replace CTM by *matrix*
-	**gsave**	-	save graphics state
-	**grestore**	-	restore graphics state

A Fibonacci Graph

```
/fibInc {                          % m n -> n (m+n)
   exch                            % m n -> n m
   1 index                         % n m -> n m n
   add
} def

/x 0 def
/y 0 def
/dx 10 def

newpath
100 100 translate                  % make (100, 100) the origin
x y moveto                         % i.e., relative to (100, 100)
0 1
25 {
   /x x dx add def                 % increment x
   dup /y exch 100 idiv def        % set y to 1/100 last fib value
   x y lineto                      % draw segment
   fibInc
} repeat

2 setlinewidth
stroke

showpage
```



Factorial

Numbers and other objects must be converted to strings before they can be printed:

int	**string**	string	create string of capacity *int*
any string	**cvs**	substring	convert to string

```
/LM 100 def                  % left margin
/FS 18 def                   % font size
/sBuf 20 string def          % string buffer of length 20

/fact {                      % n -> n!
   dup 1 lt                  % -> n bool
   { pop 1 }                 % 0 -> 1
   {
      dup                    % n -> n n
      1                      % -> n n 1
      sub                    % -> n (n-1)
      fact                   % -> n (n-1)!     NB: recursive lookup
      mul                    % n!
   }
   ifelse
} def

/showInt {                   % n -> __
   sBuf cvs show             % convert an integer to a string and show it
} def
```

```
/showFact {                       % n -> __
   dup showInt                    % show n
   (! = ) show                    % ! =
   fact showInt                   % show n!
} def

/newline {                        % __ -> __
   currentpoint exch pop          % get current y
   FS 2 add sub                   % subtract offset
   LM exch moveto                 % move to new x y
} def


/Times-Roman findfont FS scalefont setfont
LM 600 moveto

0 1 20 { showFact newline } for   % do from 0 to 20

showpage
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6.22702e+09
14! = 8.71783e+10
15! = 1.30767e+12
16! = 2.09228e+13
17! = 3.55687e+14
18! = 6.40237e+15
19! = 1.21645e+17
20! = 2.4329e+18
```

Boolean, control and string operators

any_1 any_2	**eq**	bool	test equal
any_1 any_2	**ne**	bool	test not equal
any_1 any_2	**ge**	bool	test greater or equal
-	**true**	true	push boolean value *true*
-	**false**	bool	test equal
bool proc	**if**	-	execute *proc* if *bool* is true
bool $proc_1$ $proc_2$	**ifelse**	-	execute $proc_1$ if *bool* is true else $proc_2$
init incr limit proc	**for**	-	execute *proc* with values *init* to *limit* by steps of *incr*
int proc	**repeat**	-	execute *proc* *int* times
string	**length**	int	number of elements in *string*
string index	**get**	int	get element at position *index*
string index int	**put**	-	put *int* into *string* at position *index*
string proc	**forall**	-	execute *proc* for each element of *string*

A simple formatter

```
/LM 100 def                    % left margin
/RM 250 def                    % right margin
/FS 18 def                     % font size

/showStr {                     % string -> __
   dup stringwidth pop         % get (just) string's width
   currentpoint pop            % current x position
   add                         % where printing would bring us
   RM gt { newline } if        % newline if this would overflow RM
   show
} def

/newline {                     % __ -> __
   currentpoint exch pop       % get current y
   FS 2 add sub                % subtract offset
   LM exch moveto              % move to new x y
} def

/format { { showStr ( ) show } forall } def   % array -> __

/Times-Roman findfont FS scalefont setfont
LM 600 moveto

[ (Now) (is) (the) (time) (for) (all) (good) (men) (to)
(come) (to) (the) (aid) (of) (the) (party.) ] format

showpage
```

Now is the time for
all good men to
come to the aid of
the party.

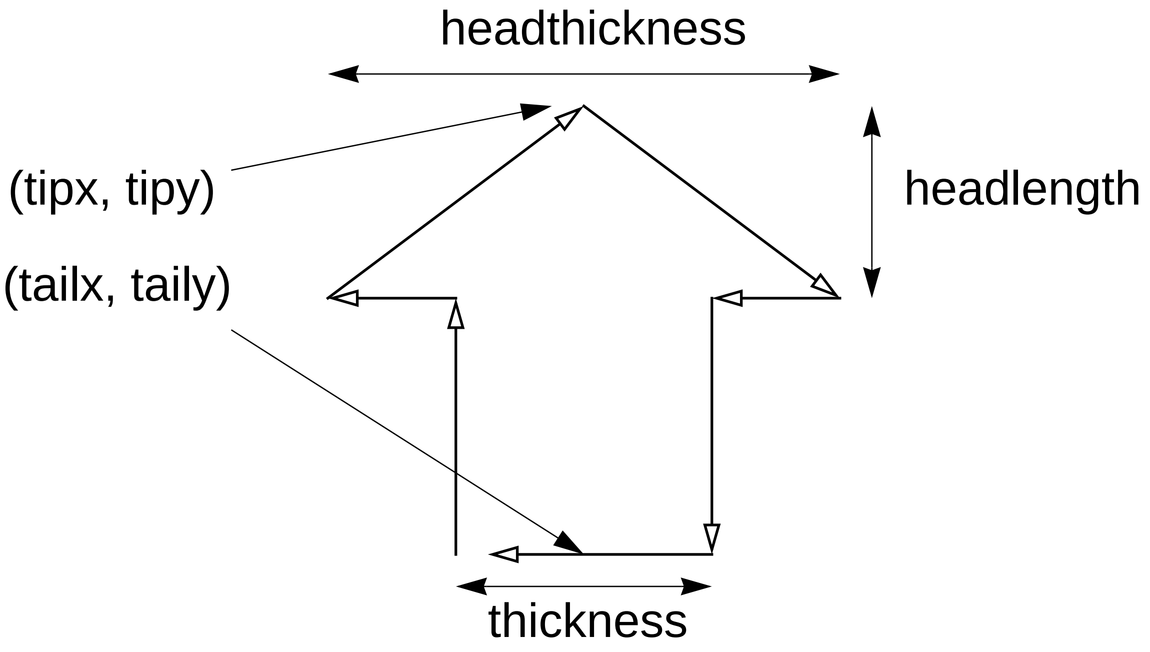
Array and dictionary operators

-	**[**	mark	start array construction
mark obj_0 ... obj_{n-1}	**]**	array	end array construction
int	**array**	array	create array of length *n*
array	**length**	int	number of elements in array
array index	**get**	any	get element at *index* position
array index any	**put**	-	put element at *index* position
array proc	**forall**	-	execute *proc* for each *array* element
int	**dict**	dict	create dictionary of capacity *int*
dict	**length**	int	number of key-value pairs
dict	**maxlength**	int	capacity
dict	**begin**	-	push *dict* on dict stack
-	**end**	-	pop dict stack

Arrowheads

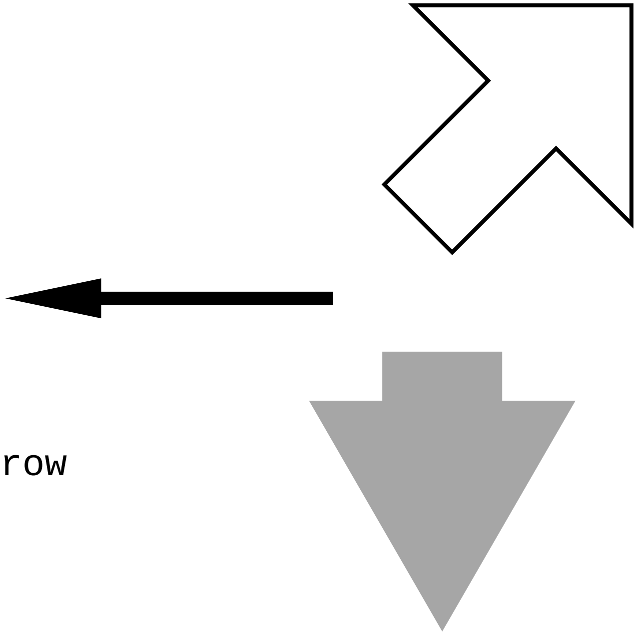
```
/arrowdict 14 dict def                  % make a new dictionary

arrowdict begin
   /mtrx matrix def                     % allocate space for a matrix
end

/arrow {
   arrowdict begin                         % open the dictionary
      /headlength exch def                 % pick up the arguments
      /halfheadthickness exch 2 div def
      /halfthickness exch 2 div def
      /tipy exch def
      /tipx exch def
      /taily exch def
      /tailx exch def

      /dx tipx tailx sub def
      /dy tipy taily sub def
      /arrowlength dx dx mul dy dy mul add sqrt def
      /angle dy dx atan def
      /base arrowlength headlength sub def

      /savematrix mtrx currentmatrix def   % save the coordinate system

      tailx taily translate                % translate to start of arrow
      angle rotate                         % rotate coordinates
```

```
        0 halfthickness neg moveto                  % draw as if starting from (0,0)
        base halfthickness neg lineto
        base halfheadthickness neg lineto
        arrowlength 0 lineto
        base halfheadthickness lineto
        base halfthickness lineto
        0 halfthickness lineto
        closepath
        savematrix setmatrix                        % restore coordinate system
      end
  } def
```



Instantiating Arrows

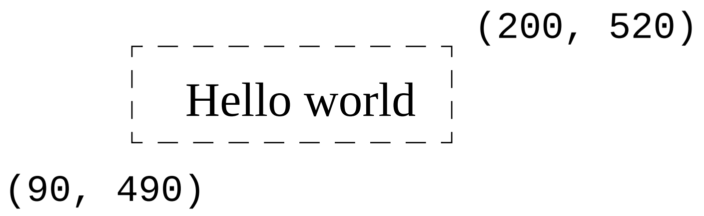
```
newpath
   318 340 72 340 10 30 72 arrow
fill

newpath
   382 400 542 560 72 232 116 arrow
3 setlinewidth stroke

newpath
   400 300 400 90 90 200 200 3 sqrt mul 2 div arrow
.65 setgray fill

showpage
```



Encapsulated PostScript

EPSF is a standard format for importing and exporting PostScript files between applications.

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 90 490 200 520
/Times-Roman findfont
        18 scalefont
        setfont
100 500 moveto
(Hello world) show
showpage
```



Summary

You should know the answers to these questions:

- What kinds of stacks does PostScript manage?
- When does PostScript push values on the operand stack?
- What is a *path*, and how can it be displayed?
- How do you manipulate the coordinate system?
- Why would you define your own dictionaries?
- How do you compute a bounding box for your PostScript graphic?

Can you answer the following questions?

- *How would you program this graphic?* 

- *When should you use* `translate` *instead of* `moveto`*?*
- *How could you use dictionaries to simulate object-oriented programming?*

3. Functional Programming

Overview

- ❑ Functional vs. Imperative Programming
- ❑ Referential Transparency
- ❑ Recursion
- ❑ Pattern Matching
- ❑ Higher Order Functions
- ❑ Lazy Lists

References:

- ❑ Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, pp 359-411.
- ❑ Paul Hudak and Joseph H. Fasel, “A Gentle Introduction to Haskell,” ACM SIGPLAN Notices, vol. 27, no. 5, May 1992, pp. T1-T53.
- ❑ J. Peterson and K. Hammond (editors). *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.4)*. Yale University, Feb. 1997

A Bit of History

Lambda Calculus (Church, 1932-33):

- ☞ formal model of computation

Lisp (McCarthy, 1960):

- ☞ symbolic computations with lists

APL (Iverson, 1962):

- ☞ algebraic programming with arrays

ISWIM (Landin, 1966):

- ☞ *let* and *where* clauses
- ☞ equational reasoning; birth of “pure” functional programming ...

ML (Edinburgh, 1979):

- ☞ originally meta language for theorem proving

SASL, KRC, Miranda (Turner, 1976-85):

- ☞ lazy evaluation

Haskell (Hudak, Wadler, et al., 1988):

- ☞ “Grand Unification” of functional languages ...

Programming without State

Imperative style:

```
n := x;
a := 1;
while n>0 do
begin a:= a*n;
   n := n-1;
end;
```

Declarative (functional) style:

```
fac n =  if    n == 0
         then  1
         else  n * fac (n-1)
```

Programs in pure functional languages have no explicit state.
Programs are constructed entirely by composing expressions.

Pure Functional Programming Languages

What is a Program?

A program (computation) is a transformation from input data to output data.

Imperative Programming:

☞ Program = Algorithms + Data

Functional Programming:

☞ Program = Functions $\circ$ Functions

Key features of pure functional languages:

1. All programs and procedures are functions
2. There are no variables or assignments — only input parameters
3. There are no loops — only recursive functions
4. The value of a function depends only on the values of its parameters
5. Functions are first-class values

Haskell

> *Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on lazy functional languages.*
>
> *— The Haskell report, version 1.4*

Referential Transparency

A function has the property of *referential transparency* if its value depends only on the values of its parameters.

✎ *Does* `f(x)+f(x)` *equal* `2*f(x)`*? In C? In Haskell?*

Referential transparency means that “equals can be replaced by equals”.

In a pure functional language, all functions are referentially transparent, and therefore always yield the same result no matter how often they are called.

Evaluation of Expressions

Expressions can be (formally) evaluated by substituting arguments for formal parameters in function bodies:

```
fac 4    ➪ if 4 == 0 then 1 else 4 * fac (4-1)
         ➪ 4 * fac (4-1)
         ➪ 4 * fac 3
         ➪ 4 * (if 3 == 0 then 1 else 3 * fac (3-1))
         ➪ 4 * 3 * fac (3-1)
         ➪ 12 * fac (3-1)
         ➪ 12 * fac 2
         ➪ 12 * (if 2 == 0 then 1 else 2 * fac (2-1))
         ➪ 12 * 2 * fac (2-1)
         ➪ 24 * fac (2-1)
         ➪ ...
         ➪ 24 * 1
         ➪ 24
```

Tail Recursion

Recursive functions can be less efficient than loops because of the high cost of procedure calls on most hardware.

- ☞ A *tail recursive function* calls itself only as its last operation, so the recursive call can be optimized away by a modern compiler.

A recursive function can be converted to a tail-recursive one by representing partial computations as explicit function parameters:

```
sfac s n = if    n == 0
           then  s
           else  sfac (s*n) (n-1)
```

```
sfac 1 4    ➪  sfac (1*4) (4-1)
            ➪  sfac 4 3
            ➪  sfac (4*3) (3-1)
            ➪  sfac 12 2
            ➪  sfac (12*2) (2-1)
            ➪  sfac 24 1
            ➪  ... ➪ 24
```

Equational Reasoning

Theorem:

For all n >= 0, `fac n` = `sfac 1 n`

Proof of theorem:

n = 0: `fac 0` = `sfac 1 0` = `1`

n > 0: Suppose `fac (n-1)` = `sfac 1 (n-1)`

`fac n` = `n * fac (n-1)`
= `n * sfac 1 (n-1)`
= `sfac n (n-1)` — *by lemma*
= `sfac 1 n`

Lemma:

For all n >= 0, `sfac s n` = `s * sfac 1 n`

Proof of lemma:

n = 0: `sfac s 0` = `s` = `s * sfac 1 0`

n > 0: Suppose `sfac s (n-1)` = `s * sfac 1 (n-1)`

`sfac s n` = `sfac (s*n) (n-1)`
= `s * n * sfac 1 (n-1)`
= `s * sfac n (n-1)`
= `s * sfac 1 n`

Pattern Matching

Languages like Haskell support a number of styles for specifying which expressions should be evaluated for different cases of arguments:

Patterns:

```
fac' 0   = 1
fac' n   = n * fac' (n-1)                -- or: fac′ (n+1) = (n+1) * fac′ n
```

Guards:

```
fac'' n  | n == 0    = 1
         | n >= 1    = n * fac'' (n-1)
```

Lists

Lists are pairs of elements and lists of elements:

- ❑ `[ ]` stands for the empty list
- ❑ `x:xs` stands for the list with `x` as the head and `xs` as the rest of the list
- ❑ `[1,2,3]` is syntactic sugar for `1:2:3:[ ]`
- ❑ `[1..n]` stands for `[1,2,3, ... n]`

Lists can be deconstructed using patterns:

```
head (x:_) = x


len [ ]     = 0
len (x:xs)  = 1 + len xs


prod [ ]    = 1
prod (x:xs)= x * prod xs


fac''' n    = prod [1..n]
```

Higher Order Functions

Higher-order functions are *first-class values* that can be composed to produce new functions.

```
map f [ ]       = [ ]
map f (x:xs)    = f x : map f xs
```

```
map fac [1..5]
        ➪ [1, 2, 6, 24, 120]
```

Anonymous functions can be written as “lambda abstractions”:

```
map (\x -> x * x) [1..10]
        ➪ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Curried functions

A *Curried function* takes its arguments one at a time, allowing it to be treated as a higher-order function.

```
plus x y    = x + y                     -- curried addition
plus 1 2    ➪  3

inc         = plus 1                    -- bind first argument of plus to 1
inc 2       ➪  3

fac = sfac 1                            -- factorial binds first argument of
        where sfac s n                  -- a curried factorial function
                |  n == 0 = s
                |  n >= 1= sfac (s*n) (n-1)
```

Curried functions are named after the logician H.B. Curry, who popularized them.

Currying

The following higher-order function takes a binary function as an argument and turns it into a curried function:

```
curry f a b    =  f (a, b)                 -- take a binary function and curry it

plus(x,y)      = x + y                     -- not a curried function
inc            = curry plus 1              -- bind first argument of plus

sfac (s, n)    =  if    n == 0             -- not a curried function
                  then  s
                  else  sfac (s*n, n-1)
fac = (curry sfac) 1                       -- bind first argument of sfac
```

Multiple Recursion

Naive recursion may result in unnecessary recalculations:

```
fib 1         = 1
fib 2         = 1
fib (n+2)     = fib n + fib (n+1)
```

Efficiency can be regained by explicitly passing calculated values:

```
fib' 1        = 1
fib' n        = a            where (a,_) = fibPair n


fibPair 1     = (1,1)
fibPair (n+2) = (a+b,a)      where (a,b) = fibPair (n+1)
```

✎ *How would you write a tail-recursive Fibbonacci function?*

Lazy Evaluation

"Lazy", or "normal-order" evaluation only evaluates expressions when they are actually needed. Clever implementation techniques (Wadsworth, 1971) allow replicated expressions to be shared, and thus avoid needless recalculations.

So:

```
sqr n = n * n
sqr (2+5) ➪ (2+5) * (2+5) ➪ 7 * 7 ➪ 49
```

Lazy evaluation allows some functions to be evaluated even if they are passed incorrect or non-terminating arguments:

```
ifTrue True x y     = x
ifTrue False x y    = y


ifTrue True 1 (5/0)
        ➪ 1
```

Lazy Lists

Lazy lists are infinite data structures whose values are generated by need:

```
from n = n : from (n+1)
```

```
take 0 _              = [ ]
take _ [ ]            = [ ]
take (n+1) (x:xs)     = x : take n xs
```

```
take 5 (from 10)
        ➪ [10, 11, 12, 13, 14]
```

NB: The lazy list (from n) has the special syntax: [n..]

```
fibs = 1 : 1 : fibgen 1 1
      where fibgen a b = (a+b) : fibgen b (a+b)
```

```
take 10 fibs
        ➪ [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

✎ *How would you re-write fibs so that (a+b) only appears once?*

Functional Programming Style

Functional programs can often be derived in a top-down fashion:

```
primes                 = 2 : primesFrom 3     -- or just: primes = primesFrom 2

primesFrom n           = p : primesFrom (p+1)
                            where p = nextPrime n

nextPrime n
   | isPrime n         = n
   | otherwise         = nextPrime (n+1)

isPrime 2              = True
isPrime n              = notdiv primes n

notdiv (k:ps) n
   | (k*k) > n         = True
   | (mod n k) == 0    = False
   | otherwise         = notdiv ps n

take 100 primes ➪ [ 2, 3, 5, 7, 11, 13, ... 523, 541 ]
```

Summary

You should know the answers to these questions:

- What is referential transparency? Why is it important?
- When is a function tail recursive? Why is this useful?
- What is a higher-order function? An anonymous function?
- What are curried functions? Why are they useful?
- How can you avoid recalculating values in a multiply recursive function?
- What is lazy evaluation?
- What are lazy lists?

Can you answer the following questions?

- *Why don't pure functional languages provide loop constructs?*
- *When would you use patterns rather than guards to specify functions?*
- *Can you build a list that contains both numbers and functions?*
- *How would you simplify* `fibs` *so that* `(a+b)` *is only called once?*
- *What kinds of applications are well-suited to functional programming?*

4. Type Systems

Overview

- ❑ What is a Type?
- ❑ Static vs. Dynamic Typing
- ❑ Kinds of Types
- ❑ Polymorphic Types
- ❑ Overloading
- ❑ User Data Types

References:

- ❑ Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.
- ❑ L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,”“ACM Computing Surveys, 17/4, Dec. 1985, pp. 471-522.
- ❑ D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

What is a Type?

Type errors:

```
? 5 + [ ]
ERROR: Type error in application
*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]
```

A type is a set of values:

- `int` = { ... -2, -1, 0, 1, 2, 3, ... }
- `bool` = { `True`, `False` }
- `Point` = { `[x=0,y=0]`, `[x=1,y=0]`, `[x=0,y=1]` ... }

A type is a partial specification of behaviour:

- `n,m:int` ⇒ `n+m` is valid, but `not(n)` is an error
- `n:int` ⇒ `n := 1`is valid, but `n := “hello world”` is an error

What kinds of specifications are interesting? Useful?

Static and Dynamic Typing

Values have *static types* defined by the programming language.

Variables and *expressions* have *dynamic types* determined by the values they assume at run-time.

A language is *statically typed* if it is always possible to determine the (static) type of an expression based on the program text alone.

A language is *strongly typed* if it is possible to ensure that every expression is *type consistent* based on the program text alone.

A language is *dynamically typed* if only *values* have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

Type consistency may be assured by (i) compile-time type-checking, (ii) type inference, or (iii) dynamic type-checking.

Kinds of Types

All programming languages provide some set of built-in types.

Most strongly-typed modern languages provide for additional user-defined types.

- ❑ **Primitive types:** booleans, integers, floats, chars ...
- ❑ **Composite types:** functions, lists, tuples ...
- ❑ **User-defined types:** enumerations, recursive types, generic types ...

The Type Completeness Principle (Watt):

> *No operation should be arbitrarily restricted in the types of values involved.*

First-class values can be evaluated, passed as arguments and used as components of composite values. Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.

Function Types

Function types allow one to deduce the types of expressions without the need to evaluate them:

```
fact :: Int -> Int
42 :: Int                        ⇒ fact 42 :: Int
```

Curried types:

```
t1 -> t2 -> ... -> tn           ≡ t1 -> (t2 -> ( ... -> tn) ...)
```

and

```
f x1 x2 ... xn                   ≡ ( ... ((f x1) x2) ... xn).
```

so:

```
(+) :: Int -> Int -> Int         ⇒ (+) 5 :: Int -> Int
```

List and Tuple Types

List Types

A list of values of type a has the type `[a]`:

```
[ 1 ] :: [ Int ]
```

NB: All of the elements in a list must be of the same type!

```
['a', 2, False]          -- this is illegal! can't be typed!
```

Tuple Types

If the expressions `x1`, `x2`, ..., `xn` have types `t1`, `t2`, ..., `tn` respectively,
then the tuple `(x1, x2, ..., xn)`has the type `(t1, t2, ..., tn)`:

```
(1, [2], 3) :: (Int, [Int], Int)
('a', False) :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

The unit type is written `()` and has a single element which is also written as `()`.

Polymorphism

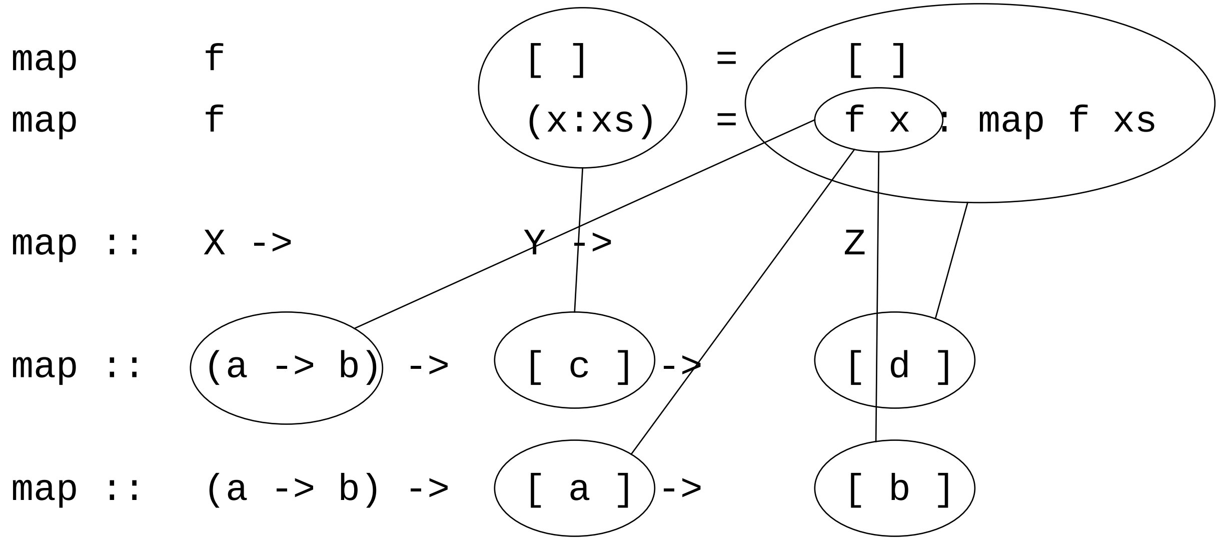
Languages like Pascal have *monomorphic type systems:* every constant, variable, parameter and function result has a unique type.

- ☞ good for type-checking
- ☞ bad for writing generic code

A *polymorphic function* accepts arguments of different types:

```
length              :: [a] -> Int
length [ ]          = 0
length (x:xs)       = 1 + length xs


map                 :: (a -> b) -> [a] -> [b]
map f [ ]           = [ ]
map f (x:xs)        = f x : map f xs


(.)                 :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x           = f (g x)
```

Polymorphic Type Inference



```
map      f              [ ]       =    [ ]
map      f              (x:xs)    =    f x : map f xs

map ::   X ->           Y ->           Z

map ::   (a -> b) ->    [ c ] ->       [ d ]

map ::   (a -> b) ->    [ a ] ->       [ b ]
```

Hindley-Milner Type Inference provides an effective algorithm for automatically determining the types of polymorphic functions. The corresponding type system is used in many modern functional languages, including ML and Haskell.

Type Specialization

A polymorphic function may be explicitly assigned a more specific type:

```
idInt :: Int -> Int
idInt x = x
```

Note that the :t command can be used to find the type of a particular expression that is inferred by Haskell:

```
? :t \x -> [x]
\x -> [x] :: a -> [a]

? :t (\x -> [x]) :: Char -> String
\x -> [x] :: Char -> String
```

Kinds of Polymorphism

Polymorphism:

- ❑ Universal:
  - – Parametric: *polymorphic map function in Haskell; nil pointer type in Pascal*
  - – Inclusion: *subtyping — graphic objects*
- ❑ Ad Hoc:
  - – Overloading: *+ applies to both integers and reals*
  - – Coercion: *integer values can be used where reals are expected and v.v.*

Coercion or overloading — how does one distinguish?

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

Overloading

Overloaded operators are introduced by means of *type classes:*

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y      = not (x == y)
```

For each overloaded instance a separate definition must be given:

```
instance Eq Int  where (==)       = primEqInt

instance Eq Bool where
   True == True                   = True
   False == False                 = True
   _ == _                         = False

instance Eq Char where c == d     = ord c == ord d

instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (u,v)                 = x==u && y==v

instance Eq a => Eq [a] where
   [ ] == [ ]                     = True
   [ ] == (y:ys)                  = False
   (x:xs) == [ ]                  = False
   (x:xs) == (y:ys)               = x==y && xs==ys
```

User Data Types

New data types can be introduced by specifying (i) a *datatype name*, (ii) a set of *parameter types*, and (iii) a set of *constructors* for elements of the type:

```
data DatatypeName a1 ... an = constr1 | ... | constrm
```

where the constructors may be:

1. Named constructors:

   ```
   Name type1 ... typek
   ```

   introduces `Name` as a new constructor of type:

   ```
   type1 -> ...-> typek -> DatatypeName a1 ... an
   ```

2. Binary constructors (i.e., starting with “:”):

   ```
   type1 CONOP type2
   ```

   introduces `(CONOP)` as a new constructor of type:

   ```
   type1 -> type2 -> DatatypeName a1 ... an
   ```

Examples of User Data Types

Enumeration types:

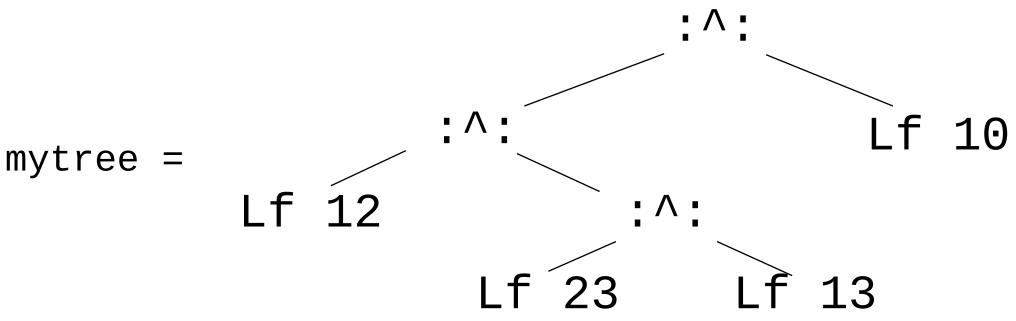
```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

whatShallIDo Sun                = “relax”
whatShallIDo Sat                = “go shopping”
whatShallIDo _                  = “looks like I'll have to go to work”
```

Union types:

```
data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp)      = temp <= 0.0
freezing (Fahrenheit temp)      = temp <= 32.0
```

Recursive Data Types

```
data Tree a = Lf a | Tree a :^: Tree a

mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```



```
? :t mytree

        ➪ mytree :: Tree Int
```

```
leaves, leaves' :: Tree a -> [a]
leaves (Lf l)       = [l]
leaves (l :^: r)    = leaves l ++ leaves r


leaves' t = leavesAcc t [ ]
   where    leavesAcc (Lf l)        = (l:)
            leavesAcc (l :^: r)     = leavesAcc l . leavesAcc r
```

Equality for Data Types and Functions

Why not automatically provide equality for all types of values?

Syntactic equality does not necessarily entail semantic equality!

User data types:

```
data Set a = Set [a]


instance Eq a => Eq (Set a) where
   Set xs == Set ys = xs `subset` ys && ys `subset` xs
      where xs `subset` ys = all (`elem` ys) xs
```

Functions:

```
? (1==) == (\x->1==x)
ERROR: Cannot derive instance in expression
*** Expression        : (==) d148 ((==) {dict} 1) (\x->(==) {dict} 1 x)
*** Required instance : Eq (Int -> Bool)
```

Summary

You should know the answers to these questions:

- How are the types of functions, lists and tuples specified?
- How can the type of an expression be inferred without evaluating it?
- What is a polymorphic function?
- How can the type of a polymorphic function be inferred?
- How does overloading differ from parametric polymorphism?
- How would you define == for tuples of length 3?
- How can you define your own data types?
- Why isn’t == pre-defined for all types?

Can you answer the following questions?

- *Can any set of values be considered a type?*
- *Why does Haskell sometimes fail to infer the type of an expression?*
- *What is the type of the predefined function* `all`*? How would you implement it?*

5. An application of Functional Programming

➪ Please review these notes in advance of the lecture!

Overview

- Huffmann encoding
  - variable length encoding based on character frequency
  - optimal encoding generation algorithm
- Architecture of a functional Huffmann encoder
- How to use recursion correctly ☞ *ensuring termination*
- Representing and manipulating trees
- Encoding trees as text; parsing stored trees
- Continuation-style IO
- “It doesn’t always pay to be lazy!” — forcing eager evaluation

References:

- H. Abelson, G. Sussman and J.Sussman, *Structure and Interpretation of Computer Programs*, MIT electrical engineering and computer science series., McGraw-Hill, 1991.

Encoding ASCII

`"I am what I am."`

Naive encoding requires at least 4 bits to encode 9 different characters:

`"`	0000
`I`	0001
(blank)	0010
`a`	0011
`m`	0100
`w`	0101
`h`	0110
`t`	0111
`.`	1000

16 characters x 4 bits/character = 64 bits

```
0000 0001 0010 0011 0100 0010 0101 0110 0011 0111 0010 0001 0010 0011 0100 0000
```

Huffmann encoding

Huffmann encoding assigns fewer bits to more frequently used characters:

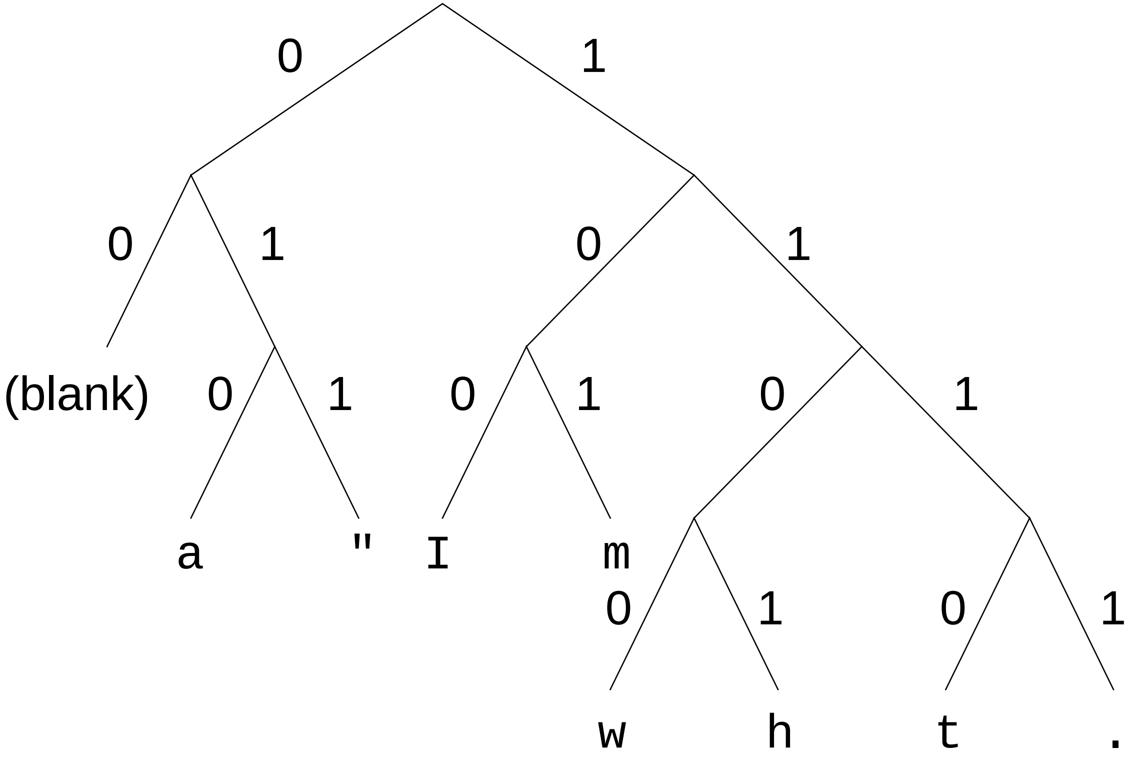
char	*frequency*	*encoding*
(blank)	4	00
`a`	3	010
`"`	2	011
`I`	2	100
`m`	2	101
`w`	1	1100
`h`	1	1101
`t`	1	1110
`.`	1	1111

$4 \times 2 + 9 \times 3 + 4 \times 4 = 51$ bits

```
011 100 00 010 101 00 1100 1101 010 1110 00 100 00 010 101 011
```

Huffmann decoding

A Huffmann encoded text can be decoded by using the bits to walk down the encoding tree and outputting the characters at the leaves:



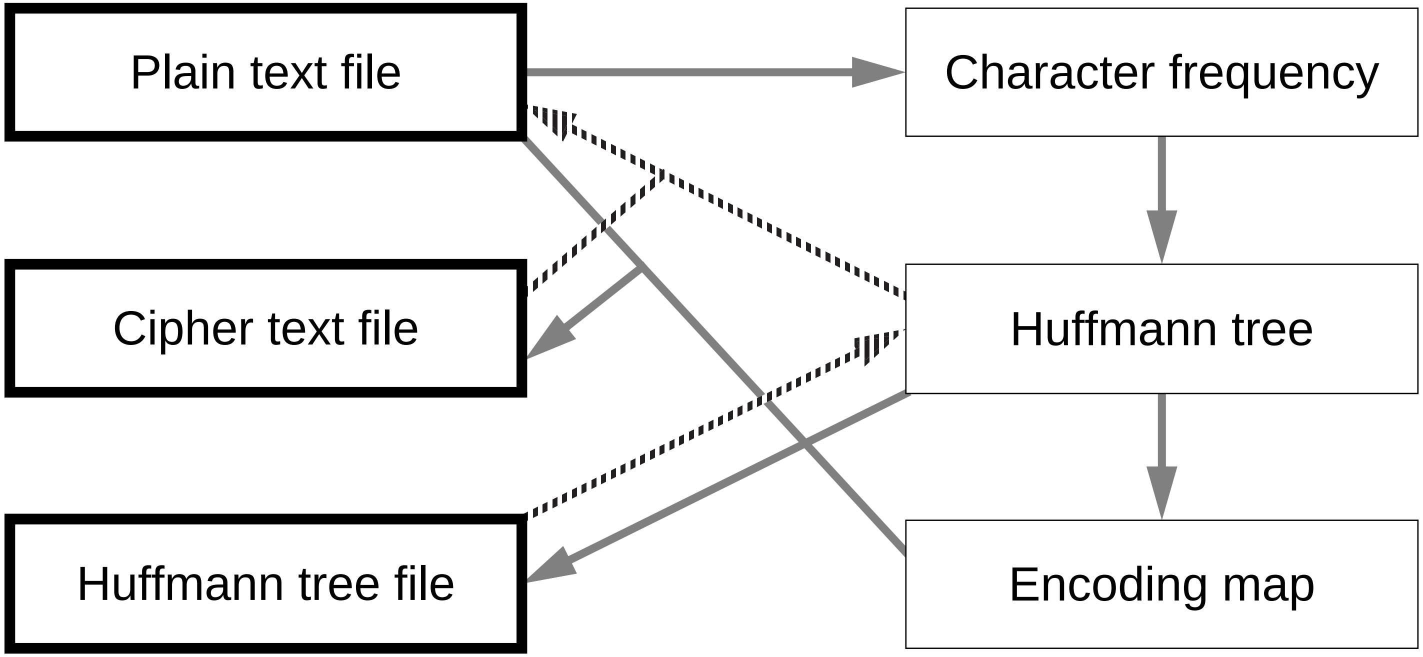
Generating optimal trees

Huffmann’s algorithm generates the optimal encoding/decoding tree by recursively merging the two “smallest” (by weight) subtrees:

- blank$_4$ a$_3$ l$_2$ m$_2$ w$_1$ h$_1$ t$_1$.$_1$
- blank$_4$ a$_3$ l$_2$ m$_2$ w$_1$ h$_1$ (t .)$_2$
- blank$_4$ a$_3$ l$_2$ m$_2$ (w h)$_2$ (t .)$_2$
- blank$_4$ a$_3$ l$_2$ m$_2$ ((w h) (t .))$_4$
- blank$_4$ a$_3$ (l m)$_4$ ((w h) (t .))$_4$
- (blank a)$_7$ (l m)$_4$ ((w h) (t .))$_4$
- (blank a)$_7$ ((l m) ((w h) (t .)))$_8$
- ((blank a) ((l m) ((w h) (t .))))$_{15}$

✎ *Write a program to Huffmann encode and decode text files.*

Architecture



Plain text file

Character frequency

Cipher text file

Huffmann tree

Huffmann tree file

Encoding map

Frequency Counting

We can represent frequency counts as lists of pairs of Chars and Ints:

```
-- Each Char appears Int (>0) times in some text
type CharCount              = (Char,Int)


-- Compute a [CharCount] for a given String
freqCount :: String -> [CharCount]
freqCount ""                = []
freqCount (c:s)             = incCount c (freqCount s)


-- Increment the [CharCount] for a given Char
incCount :: Char -> [CharCount] -> [CharCount]
incCount c []               = [(c,1)]
incCount c ((c1,n):ccList)
   | c == c1                = (c1,n+1):ccList
   | otherwise              = (c1,n):(incCount c ccList)
```

So:

```
iam = "\"I am what I am.\""
freqCount iam            ➪ [('"',2), ('.',1), ('m',2), ('a',3), (' ',4),
                            ('I',2), ('t',1), ('h',1), ('w',1)]
```

How to use recursion correctly!

In order to ensure that a recursive function will terminate:

1. Carefully establish the base cases:

```
freqCount ""                  = []
```

☞ base case is an empty string

2. Ensure that every recursive invocation *reduces some measure of size*, and therefore will eventually reach a base case

```
freqCount (c:s)               = incCount c (freqCount s)
```

☞ recursive call reduces *length of argument string* ⇒ will reach base case

Trees

We can represent a Huffmann tree as a user data type:

```
data Tree a                     = Leaf a
                                | Tree a :^: Tree a

-- Weigh a Tree
weight :: Tree CharCount -> Int
weight (Leaf (ch,n))            = n
weight (tree1 :^: tree2)        = (weight tree1) + (weight tree2)
```

Constructors are functions too:

```
map Leaf (freqCount iam) ➪ [  Leaf ('"',2), Leaf ('.',1), Leaf ('m',2),
                              Leaf ('a',3), Leaf (' ',4), Leaf ('I',2),
                              Leaf ('t',1), Leaf ('h',1), Leaf ('w',1) ]

map weight (map Leaf (freqCount iam))
                         ➪ [  2, 1, 2, 3, 4, 2, 1, 1, 1 ]
```

Merging trees

We can decompose tree merging by means of a helper function:

```
-- Recursively merge smallest trees together till a single tree results
mergeTrees :: [Tree CharCount] -> Tree CharCount
mergeTrees [tree]          = tree          -- base case: already a single tree
mergeTrees (tree1:tree2:treeList)          -- otherwise
   | w1 < w2               = mt treeList tree1 tree2 []
   | otherwise             = mt treeList tree2 tree1 []
        where { w1 = (weight tree1); w2 = (weight tree2) }


-- Usage: mt untested tr1 tr2 tested, where weight(tr1) < weight(tr2) and
-- tested is a list of trees with weights bigger than either tr1 or tr2

mt [] tr1 tr2 []           = tr1 :^: tr2
mt [] tr1 tr2 tested       = mergeTrees ((tr1 :^: tr2):tested)
mt (tr3:untested) tr1 tr2 tested
   | w3 < w1               = mt untested tr3 tr1 (tr2:tested)
   | w3 < w2               = mt untested tr1 tr3 (tr2:tested)
   | otherwise             = mt untested tr1 tr2 (tr3:tested)
        where { w1 = (weight tr1); w2 = (weight tr2); w3 = (weight tr3) }
```

✎ *Is there a more efficient way to merge trees?*

Tree merging ...

```
mergeTrees (map Leaf (freqCount iam))
   ➪  (  (  Leaf ('m',2)
             :^:
             ( Leaf ('w',1) :^: Leaf ('h',1) )
          )
          :^:
          (  ( Leaf ('.',1) :^: Leaf ('t',1) )
             :^:
             Leaf ('"',2)
          )
       )
       :^:
       (  Leaf (' ',4)
          :^:
          ( Leaf ('I',2) :^: Leaf ('a',3) )
       )
```

Extracting the Huffmann tree

We remove the character counts to leave the Huffmann tree:

```
-- Strip out the character counts from a Tree of CharCounts
charTree :: Tree CharCount -> Tree Char
charTree (Leaf (ch,n))    = Leaf ch
charTree (tr1 :^: tr2)    = (charTree tr1) :^: (charTree tr2)


-- Generate an optimal Huffmann encoding tree for a piece of text
huf :: String -> Tree Char
huf text = charTree (mergeTrees (map Leaf (freqCount text)))
```

```
huf iam ➪  (  (  Leaf 'm'
                 :^: ( Leaf 'w' :^: Leaf 'h'))
              :^: (( Leaf '.' :^: Leaf 't')
                 :^: Leaf '"' ) )
           :^: ( Leaf ' '
              :^:
              ( Leaf 'I' :^: Leaf 'a'))
```

NB: The resulting tree is not necessarily unique.

Extracting the encoding map

To encode text, we need to store the path to each Char in the tree:

```
-- From a Huffmann tree, generate the encoding map
mkEncode :: String -> (Tree Char) -> [(Char, String)]
-- remember the path to this char
mkEncode prefix (Leaf ch)       = [(ch, prefix)]

-- walk the tree, remembering which path is taken
mkEncode prefix (tr1 :^: tr2)   = (mkEncode (prefix ++ "0") tr1) ++
                                      (mkEncode (prefix ++ "1") tr2)
```

```
mkEncode "" (huf iam)
   ➪  [('m',"000"), ('w',"0010"), ('h',"0011"), ('.',"0100"), ('t',"0101"),
      ('"',"011"), (' ',"10"), ('I',"110"), ('a',"111")]
```

Applying the encoding map

To encode text, we just look up characters in the encoding map:

```
-- lookup a char in an encoding map
encChar :: [(Char, String)] -> Char -> String
encChar [] _     = undefined                  -- should never happen!
encChar ((ch,str):table) c
   | c == ch     = str
   | otherwise   = encChar table c
```

```
encode :: Tree Char -> String -> String
encode tree text = foldr (++) "" (map (encChar (mkEncode "" tree)) text)
```

```
encode (huf iam) iam ⇨ 011110101110001000100011111010110110101110000100011
```

NB: `foldr` is defined in the standard prelude:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Decoding by walking the tree

To decode text, we just walk the tree, keeping a copy of the original tree so we can start over from the root each time we reach a leaf:

```
decode :: Tree Char -> String -> String
decode tree = walk tree tree                    -- NB: higher order

walk :: Tree Char -> Tree Char -> String -> String
walk tree (tr1 :^: tr2) ('0':rest)    = walk tree tr1 rest
walk tree (tr1 :^: tr2) ('1':rest)    = walk tree tr2 rest
walk tree (Leaf ch) rest              = [ch] ++ walk tree tree rest
walk tree nav []                      = []
```

```
decode (huf iam) (encode (huf iam) iam) ➪ "\"I am what I am.\""
```

Representing trees as text

We need a way to store Huffmann trees as plain text.

We represent leaves by their character values, and intermediate nodes as parenthesized expressions, but we must take care to encode parentheses:

```
-- Show a Tree Char as a Lisp-style parenthesized string
showTree :: Tree Char -> String
showTree (Leaf ch)
   | ch == '('              = "\\("
   | ch == ')'              = "\\)"
   | ch == '\\'             = "\\\\"
   | ch == '\n'             = "\\n"
   | otherwise              = [ch]
showTree (tr1 :^: tr2)     = "(" ++ (showTree tr1) ++ (showTree tr2) ++ ")"
```

```
showTree (huf iam)                ➪ (((m(wh))((.t)"))( (Ia)))

showTree (huf "()\\\n")           ➪ "((\\\\\\n)(\\(\\)))"
putStr (showTree (huf "()\\\n"))  ➪  ((\\\n)(\(\)))
```

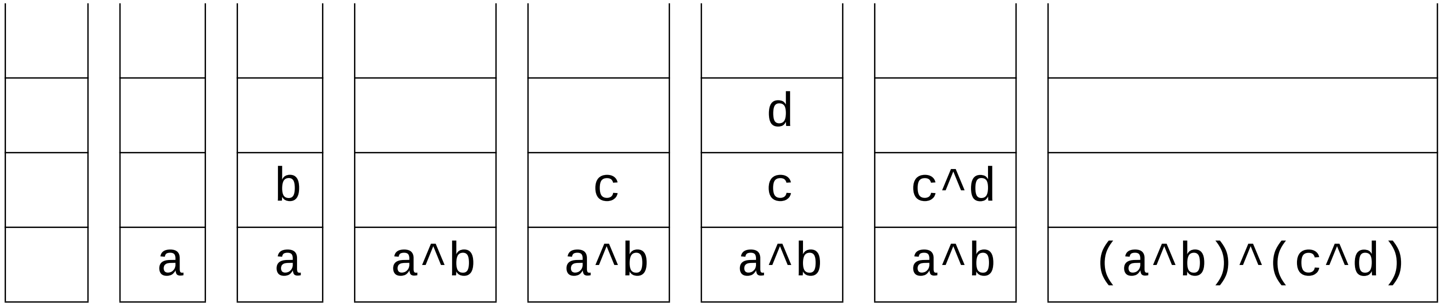
Using a stack to parse stored trees

Naturally, we need a way to parse and reconstruct the stored trees.

A standard solution is to push the leaves on a stack of trees, joining the top two elements every time a right parenthesis is encountered:

Example:

```
((ab)(cd))
```



If the parentheses are balanced, a single tree will be left on the stack.

Parsing stored trees

```
-- Parse a Lisp-style parenthesized string, generating a Tree Char
parseTree :: String -> Tree Char
parseTree                = pt []

pt :: [Tree Char] -> String -> Tree Char
pt [tree] []             = tree
pt stack (ch:str)
   | ch == '('           = pt stack str
   | ch == ')'           = pt (join stack) str
   | ch == '\\'          = pt (Leaf (unescape (head str)):stack) (tail str)
   | otherwise           = pt (Leaf ch:stack) str

-- join the top two trees of the stack into one
join :: [Tree a] -> [Tree a]
join (tr1:tr2:stack)     = (tr2:^:tr1):stack

-- unescape the character following a backslash
unescape :: Char -> Char
unescape '('             = '('
unescape ')'             = ')'
unescape '\\'            = '\\'
unescape 'n'             = '\n'
```

```
parseTree (showTree (huf "()\\\n"))
   ➪ (Leaf '\' :^: Leaf '\n') :^: (Leaf '(' :^: Leaf ')')
```

IO in a stateless world

If “pure functional” languages were truly “pure”, IO would be impossible, since reading and writing files are side effects. Haskell gets around this by pretending that files are lazy lists of characters:

```
type FilePath = [Char]
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
catch :: IO a -> (IOError -> IO a) -> IO a
```

So we can write:

```
import IO

-- copy file f1 to f2
fcopy :: FilePath -> FilePath -> IO ()
fcopy f1 f2 = catch fcopy' (ioerr ("Can't open " ++ f1)) where
   fcopy' = do              -- “do” takes a sequence of “statements” as arguments!
      s <- readFile f1
      catch (fcopy'' s) (ioerr ("Can't write " ++ f2)) where
         fcopy'' x = do writeFile f2 x

-- print an error message
ioerr :: String -> IOError -> IO ()
ioerr message _ = do hPrint stderr message
```

Encoding files

Now we have all the pieces to Huffmann encode plain text files:

```
-- reads a plain text file and generates the cipher and tree files
enc :: FilePath -> IO ()
enc plain = let {
      treefile    = plain ++ ".huf" ;
      cipher      = plain ++ ".enc"
   } in
   catch (enc' plain treefile cipher)
      (\e -> putStr ("Error: " ++ show e)) where
   enc' plain treefile cipher =
      do contents <- readFile plain
         tree <- return (huf contents)
         writeFile treefile (showTree tree)
         writeFile cipher (encode tree contents)
         putStr "Done."
```

Decoding Files

Decoding is similar:

```
dec :: FilePath -> IO()
dec plain =
   let {
      treefile    = plain ++ ".huf";
      cipher      = plain ++ ".enc"
   } in
   catch (dec' plain treefile cipher)
      (\e -> putStr ("Error: " ++ show e)) where
   dec' plain treefile cipher =
      do contents <- readFile treefile
         tree <- return (parseTree contents)
         text <- readFile cipher
         writeFile plain (decode tree text)
         putStr "Done."
```

See chapter 7 of “A Gentle Introduction to Haskell” for the complete story on IO!

Testing the program

From shell:

```
echo '"I am what I am."' > iam
```

From Haskell:

```
enc "iam"
```

From shell:

```
% cat iam.huf
➪ ((((\n.)(wh)) )((mI)((t")a)))

% cat iam.enc
➪ 110110101111100010010001111111000110101111100000111010000
```

From Haskell:

```
enc "huf"
➪ (5339 reductions, 16064 cells)
  ERROR: Control stack overflow
```

Tracing our program

Because Haskell is a "lazy" language, no expression is evaluated until it is actually needed:

```
freqCount "abc"
>>>> freqCount "abc"
===> incCount 'a' (freqCount "bc")
===> incCount 'a' (incCount 'b' (freqCount "c"))
===> incCount 'a' (incCount 'b' (incCount 'c' (freqCount "")))
===> incCount 'a' (incCount 'b' (incCount 'c' []))
===> incCount 'a' (incCount 'b' (('c',1) : []))
===> incCount 'a' (('c',1) : incCount 'b' [])
===> ('c',1) : incCount 'a' (incCount 'b' [])
===> ('c',1) : incCount 'a' (('b',1) : [])
===> ('c',1) : ('b',1) : incCount 'a' []
===> ('c',1) : ('b',1) : ('a',1) : []
(26 reductions, 97 cells)
```

Although the frequency count list will have a maximum size of 256 (for 256 ASCI chars), nothing will be evaluated until *the entire file has been read!*

Frequency Counting Revisited

We need frequency counting to be evaluated eagerly!

We can *force* evaluation by requiring values to be produced

```
-- eager, tail-recursive frequency counter
-- fcEager (c:s) front back -- front does not contain c, back to be checked
fcEager :: String -> [CharCount] -> [CharCount] -> [CharCount]
fcEager "" [] ccl                 = ccl
fcEager (c:s) front []            = fcEager s [] ((c,1):front)
fcEager (c:s) front ((c1,n):back)
   | (c == c1)                    = fcEager s [] (front ++ ((c,n+1):back))
   | otherwise                    = fcEager (c:s) ((c1,n):front) back


fc2 s = fcEager s [] []           -- replaces original freqCount

enc2 = ...


enc2 "huf"                        -- encode this program (9334 bytes)
   ➪ (2117457 reductions, 6145824 cells, 100 garbage collections)
```

Summary

You should know the answers to these questions:

- How can you be sure a recursive function will terminate?
- How do you know where characters end in Huffmann encoded bit strings?
- How can you generate a tree from its string representation?
- Why doesn’t Haskell have to load the entire file into memory when readFile is evaluated?

Can you answer the following questions?

- *Can you prove that Huffmann’s algorithm really generates the optimal map?*
- *What would happen if* `encode` *used* `foldl` *instead of* `foldr`*?*
- *Can parseTree be re-written so it uses the run-time stack instead of representing a stack as a list?*
- *Our Huffmann encoder actually outputs one byte for each “0” or “1”!*
  How would you adapt the program to produce bits instead of bytes?

6. Introduction to the Lambda Calculus

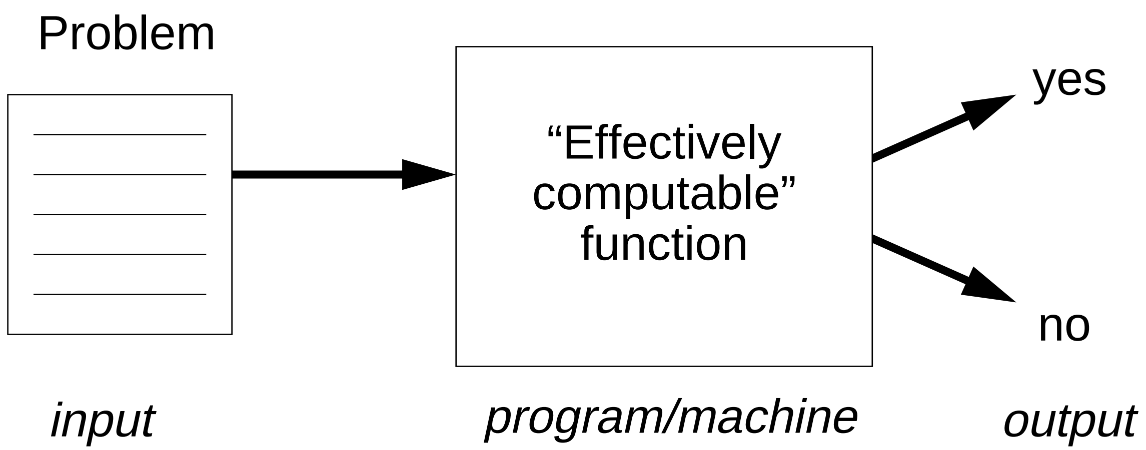
Overview

- ❑ What is Computability? — Church's Thesis
- ❑ Lambda Calculus — operational semantics
- ❑ The Church-Rosser Property
- ❑ Modelling basic programming constructs

References:

- ❑ Paul Hudak, "Conception, Evolution, and Application of Functional Programming Languages," ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.
- ❑ Kenneth C. Louden, *Programming Languages: Principles and Practice*, PWS Publishing (Boston), 1993.
- ❑ H.P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland, 1984, Revised edition.

What is Computable?



Computation is usually modelled as a mapping from inputs to outputs, carried out by a formal “machine,” or program, which processes its input in a sequence of steps.

An “effectively computable” function is one that can be computed in a finite amount of time using finite resources.

Church's Thesis

> *Effectively computable functions [from positive integers to positive integers] are just those definable in the lambda calculus.*

Or, equivalently:

> *It is not possible to build a machine that is more powerful than a Turing machine.*

Church's thesis cannot be proven because "effectively computable" is an intuitive notion, not a mathematical one. It can only be refuted by giving a counter-example — a machine that can solve a problem not computable by a Turing machine.

So far, *all* models of effectively computable functions have shown to be equivalent to Turing machines (or the lambda calculus).

Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called *uncomputable*.

☞ Assuming Church's thesis is true, an uncomputable problem cannot be solved by *any* real computer.

The Halting Problem

> *Given an arbitrary Turing machine and its input tape, will the machine eventually halt?*

The Halting Problem is provably uncomputable — which means that it cannot be solved in practice.

What is a Function?

Extensional view:

A (total) *function* f: $A \rightarrow B$ is a subset of $A \times B$ (i.e., a *relation*) such that:

1. for each $a \in A$, there exists some $(a,b) \in f$ (i.e., f(a) is defined), and
2. if $(a,b_1) \in f$ and $(a, b_2) \in f$, then $b_1 = b_2$ (i.e., f(a) is unique)

Intensional view:

A *function* f: $A \rightarrow B$ is an *abstraction* $\lambda\, x\, .\, e$,
where x is a variable name,
and e is an expression,
such that when a value $a \in A$ is substituted for x in e,
then this expression (i.e., f(a)) evaluates to some (unique) value $b \in B$.

The (Untyped) Lambda Calculus

The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.

Syntax:

e	$::=$	x	*a variable*
	$\vert$	$\lambda\, x\, .\, e$	*an abstraction (function)*
	$\vert$	$e_1\; e_2$	*a (function) application*

(Operational) Semantics:

α conversion (renaming):	$\lambda\, x\, .\, e$	$\leftrightarrow \lambda\, y\, .\, [\, y/x\,]\, e$	*where y is not free in e*
β reduction (application):	$(\lambda\, x\, .\, e_1)\; e_2$	$\rightarrow [\, e_2/x\,]\, e_1$	*avoiding name capture*
η reduction:	$\lambda\, x\, .\, (e\; x)$	$\rightarrow e$	*if x is not free in e*

The lambda calculus can be viewed as the simplest possible pure functional programming language.

Beta Reduction

Beta reduction is the computational engine of the lambda calculus:

Define: $I \equiv \lambda\, x\, .\, x$

Now consider:

$$
\begin{array}{llll}
I\ I = (\lambda\, x\, .\, x)\ (\lambda\, x\, .\, x\,) & \rightarrow & [\,(\lambda\, x\, .\, x\,)\ /\ x]\ x & \beta \textit{ reduction} \\
 & = & (\lambda\, x\, .\, x) & \textit{substitution} \\
 & = & I &
\end{array}
$$

We can implement most lambda expressions directly in Haskell:

```
i = \x -> x

? i 5
5
(2 reductions, 6 cells)
? i i 5
5
(3 reductions, 7 cells)
```

Free and Bound Variables

The variable x is *bound* by the enclosing λ in the expression: $\lambda\, x.e$

A variable that is not bound, is *free* :

$$
\begin{aligned}
fv(x) &= \{ x \} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\lambda\, x\, .\, e) &= fv(e) - \{ x \}
\end{aligned}
$$

An expression with no free variables is *closed* (otherwise it is *open*)

For example, y is bound and x is free in the (open) expression: $\lambda\, y\, .\, x\, y$

Syntactic substitution will not work:

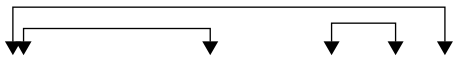
$(\lambda\, x\, .\, \lambda\, y\, .\, x\, y\,)\, y$	$\rightarrow$	$[\, y / x]\, (\lambda\, y\, .\, x\, y)$	β *reduction*
	$\neq$	$(\lambda\, y\, .\, y\, y\,)$	*incorrect substitution!*

Since y is already bound in (λ y . x y), we *cannot* directly substitute y for x.

Substitution

We must define substitution carefully to avoid name capture:

$$
\begin{aligned}
[e/x]\ x &= e && \\
[e/x]\ y &= y && \textit{if } x \neq y \\
[e/x]\ (e_1\ e_2) &= ([e/x]\ e_1)\ ([e/x]\ e_2) && \\
[e/x]\ (\lambda\ x\ .\ e_1) &= (\lambda\ x\ .\ e_1) && \\
[e/x]\ (\lambda\ y\ .\ e_1) &= (\lambda\ y\ .\ [e/x]\ e_1) && \textit{if } x \neq y \textit{ and } y \notin fv(e) \\
[e/x]\ (\lambda\ y\ .\ e_1) &= (\lambda\ z\ .\ [e/x]\ [z/y]\ e_1) && \textit{if } x \neq y \textit{ and } z \notin fv(e) \cup fv(e_1)
\end{aligned}
$$



Consider:

$$
\begin{aligned}
(\ \lambda\ x\ .\ (\ \lambda\ y\ .\ x\ )\ (\lambda\ x\ .\ x)\ x\ )\ y &\rightarrow [y\ /\ x]\ (\ \lambda\ y\ .\ x\ )\ (\lambda\ x\ .\ x)\ x\ ) \\
&= (\ \lambda\ z\ .\ y\ )\ (\lambda\ x\ .\ x)\ y\ )
\end{aligned}
$$

Alpha Conversion

Alpha conversions allows one to rename bound variables.

A bound name x in the lambda abstraction (λ x.e) may be substituted by any other name y, *as long as there are no free occurrences of y in e:*

Consider:

$(\lambda x . \lambda y . x y) y$	$\rightarrow$	$(\lambda x . \lambda z . x z) y$	α *conversion*
	$\rightarrow$	$[y / x] (\lambda z . x z)$	β *reduction*
	$=$	$(\lambda z . y z)$	

Eta Reduction

Eta reductions allows one to remove “redundant lambdas”.

Suppose that f is a closed expression (i.e., x does not occur free in f).
Then:

$(\lambda x . f x) y$	$\rightarrow$	$f y$	β *reduction*

More generally, this will hold whenever x does not occur free in f.
In such cases, we can always rewrite $(\lambda x . f x)$ as f.

Normal Forms

A lambda expression is in *normal form* if it can no longer be reduced by the beta or eta reduction rules.

Not all lambda expressions have normal forms!

$$\begin{array}{llll}
\Omega = (\lambda x . x x) (\lambda x . x x) & \rightarrow & [(\lambda x . x x) / x] (x x) & \\
 & = & (\lambda x . x x) (\lambda x . x x) & \beta \textit{ reduction} \\
 & \rightarrow & (\lambda x . x x) (\lambda x . x x) & \beta \textit{ reduction} \\
 & \rightarrow & (\lambda x . x x) (\lambda x . x x) & \beta \textit{ reduction} \\
 & \rightarrow & \ldots &
\end{array}$$

Reduction of a lambda expression to a normal form is analogous to a Turing machine halting or a program terminating.

Evaluation Order

Most programming languages are *strict*, that is, all expressions passed to a function call are evaluated before control is passed to the function.

Most modern functional languages, on the other hand, use *lazy* evaluation, that is, expressions are only evaluated when they are needed.

Consider:

```
sqr n = n * n
```

Applicative-order reduction:

```
sqr (2+5) ➪ sqr 7 ➪ 7*7 ➪ 49
```

Normal-order reduction:

```
sqr (2+5) ➪ (2+5) * (2+5) ➪ 7 * (2+5) ➪ 7 * 7 ➪ 49
```

The Church-Rosser Property

> *“If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative order reduction), then all of these evaluation orders yield the same result”.*

So, evaluation order “does not matter” in the lambda calculus.

However , applicative order reduction may not terminate, even if a normal form exists!

$$(\lambda x . y)((\lambda x . x x)(\lambda x . x x))$$

Applicative order reduction	Normal order reduction
$\rightarrow (\lambda x . y)((\lambda x . x x)(\lambda x . x x))$	$\rightarrow y$
$\rightarrow (\lambda x . y)((\lambda x . x x)(\lambda x . x x))$	
$\rightarrow \ldots$	

Currying

Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modelled as *curried* higher-order functions [named after the logician H.B. Curry, who popularized the approach].

To improve readability, multiple lambdas can be suppressed, so:

$$\lambda\, x\, y\, .\, x \quad = \quad \lambda\, x\, .\, \lambda\, y\, .\, x$$

$$\lambda\, b\, x\, y\, .\, b\, x\, y \quad = \quad \lambda\, b\, .\, \lambda\, x\, .\, \lambda\, y\, .\, b\, x\, y$$

Representing Booleans

Although the lambda calculus is extremely sparse, most (sequential) programming constructs can be built up as lambda expressions.

Define:

$\text{True} \equiv \lambda\, x\ y\ .\ x$

$\text{False} \equiv \lambda\, x\ y\ .\ y$

$\text{not} \equiv \lambda\, b\ .\ b\ \text{False True}$

$\text{if } b \text{ then } x \text{ else } y \equiv \lambda\, b\ x\ y\ .\ b\ x\ y$

Then:

$\text{not True} = (\ \lambda\, b\ .\ b\ \text{False True}\ )\ (\ \lambda\, x\ y\ .\ x\ )$

$\rightarrow (\ \lambda\, x\ y\ .\ x\ )\ \text{False True}$

$\rightarrow \text{False}$

$\text{if True then } x \text{ else } y = (\ \lambda\, b\ x\ y\ .\ b\ x\ y\ )\ (\lambda\, x\ y\ .\ x)\ x\ y$

$\rightarrow (\lambda\, x\ y\ .\ x)\ x\ y$

$\rightarrow x$

Representing Tuples

Although tuples are not supported by the lambda calculus, they can easily be modelled as higher-order functions that “wrap” pairs of values.

n-tuples can be modelled by composing pairs ...

Define:

pair $\equiv$ $(\lambda\, x\, y\, z\, .\, z\, x\, y)$

first $\equiv$ $(\lambda\, p\, .\, p \text{ True })$

second $\equiv$ $(\lambda\, p\, .\, p \text{ False })$

Then:

$$(1, 2) = \text{pair } 1\ 2$$
$$\rightarrow (\lambda\, z\, .\, z\ 1\ 2)$$

In Haskell:

```
t       = \x -> \y -> x
f       = \x -> \y -> y
pair    = \x -> \y -> \z -> z x y
first   = \p -> p t
second  = \p -> p f
```

```
? first (pair 1 2)
1
? first (second (pair 1 (pair 2 3)))
2
```

Representing Numbers

There is a “standard encoding” of natural numbers into the lambda calculus:

Define:

$0 \equiv (\lambda x . x)$

$\text{succ} \equiv (\lambda n . (\text{False}, n))$

So:

$1 \equiv \text{succ } 0 \rightarrow (\text{False}, 0)$

$2 \equiv \text{succ } 1 \rightarrow (\text{False}, 1)$

Consider:

$\text{iszero} \equiv \text{first}$

$\text{pred} \equiv \text{second}$

Then:

$\text{iszero } 1 = \text{first } (\text{False}, 0) \rightarrow \text{False}$

$\text{iszero } 0 = (\lambda p . p \text{ True}) (\lambda x . x) \rightarrow \text{True}$

$\text{pred } 1 = \text{second } (\text{False}, 0) \rightarrow 0$

Summary

You should know the answers to these questions:

- ❑ Is it possible to write a Pascal compiler that will generate code just for programs that terminate?
- ❑ What are the alpha, beta and eta conversion rules?
- ❑ What is name capture? How does the lambda calculus avoid it?
- ❑ What is a normal form? How does one reach it?
- ❑ How can Booleans, tuples and numbers be represented in the lambda calculus?

Can you answer the following questions?

- ✎ *How can name capture occur in a programming language?*
- ✎ *What happens if you try to program* Ω *in Haskell? Why?*
- ✎ *What do you get when you try to evaluate (pred 0)? What does this mean?*
- ✎ *How would you model negative integers in the lambda calculus? Fractions?*
- ✎ *Is it possible to model real numbers? Why, or why not?*

7. Fixed Points

Overview

- ❑ Recursion and the Fixed-Point Combinator
- ❑ The typed lambda calculus
- ❑ The polymorphic lambda calculus
- ❑ A quick look at process caculi

References:

- ❑ Paul Hudak, “Conception, Evolution, and Application of Functional Programming Languages,” ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.

Recursion

Suppose we want to define arithmetic operations on our lambda-encoded numbers.

In Haskell we can program:

```
plus n m
   | n == 0         = m
   | otherwise      = plus (n-1) (m+1)
```

so we might try to define:

$$\text{plus} \equiv \lambda\, n\ m\ .\ \text{iszero}\ n\ m\ (\ \text{plus}\ (\ \text{pred}\ n\ )\ (\ \text{succ}\ m\ )\ )$$

Unfortunately this is *not* a definition, since we are trying to use plus before it is defined.

Although recursion is fundamental to functional programming, it is not primitive in the lambda calculus, so we must find a way to “program” it!

The Fixed Point Combinator

We can obtain a closed expression by abstracting over plus:

$$\begin{array}{lll} \text{rplus} & \equiv & \lambda\ \text{plus n m . iszero n} \\ & & \qquad\qquad \text{m} \\ & & \qquad\qquad \text{(plus (pred n) (succ m))} \end{array}$$

Let fplus be the actual addition function we want. We must pass it to rplus as a parameter before we can perform any additions. But then (rplus fplus) is the function we want.

In other words, we are looking for an fplus such that:

$$\text{rplus fplus} \leftrightarrow \text{fplus}$$

Now consider:

$$\begin{array}{lll} \text{Y} & \equiv & \lambda\ \text{f . }(\lambda\ \text{x . f (x x)) }(\lambda\ \text{x . f (x x))} \\ \text{fplus} & \equiv & \text{Y rplus} \\ & \to & (\lambda\ \text{x . rplus (x x)) }(\lambda\ \text{x . rplus (x x))} \\ & \to & \text{rplus (}(\lambda\ \text{x . rplus (x x)) }(\lambda\ \text{x . rplus (x x)))} \\ & = & \text{rplus fplus} \end{array}$$

Fixed Points

A *Fixed Point* of a function `f` is a value p such that `f p = p`.

Examples:

```
fact 1 = 1
fact 2 = 2
fib 0 = 0
fib 1 = 1
```

Fixed points are not always “well-behaved”:

```
succ n = n + 1
```

✎ *What is a fixed point of* `succ`*?*

Fixed Point Theorem

Fixed point Theorem:

Every lambda expression e has a *fixed point* p such that (e p) ↔ p.

Proof:

Let

$$Y \quad \equiv \quad \lambda\, f \,.\, (\lambda\, x \,.\, f\,(x\ x))\ (\lambda\, x \,.\, f\,(x\ x))$$

Now consider:

$$p \equiv Y\ e \quad \rightarrow \quad (\lambda\, \underline{x} \,.\, e\,(x\ x))\ \underline{(\lambda\, x \,.\, e\,(x\ x))}$$

$$\rightarrow \quad e\ \underline{((\lambda\, x \,.\, e\,(x\ x))\ (\lambda\, x \,.\, e\,(x\ x)))}$$

$$\rightarrow \quad e\ p$$

So, the “magical Y combinator” can always be used to find a fixed point of an arbitrary lambda expression.

Using the Y Combinator

Consider

$$f \equiv \lambda\, x.\ \text{True}$$

Then

$$\begin{aligned} Y\ f &\rightarrow f\ (Y\ f) \\ &= (\lambda\, x.\ \text{True})\ (Y\ f) \\ &\rightarrow \text{True} \end{aligned}$$

Consider

$$\begin{aligned} Y\ \text{succ} &\rightarrow \text{succ}\ (Y\ \text{succ}) \\ &\rightarrow (\text{False}, (Y\ \text{succ})) \end{aligned}$$

✎ *What are succ and pred of (False, (Y succ))? What does this represent?*

Recursive Functions are Fixed Points

We *cannot* write:

plus ≡ λ n m . iszero n
m
(**plus** (pred n) (succ m))

because plus is unbound in the “definition”.

We can, however, *abstract* over plus:

rplus ≡ λ plus n m . iszero n
m
(plus (pred n) (succ m))

Now we seek a lambda expression plus, such that:

rplus plus ↔ plus

I.e., *plus is a fixed point of rplus*. By the fixed point theorem, we can take:

plus ≡ Y rplus

Unfolding Recursive Lambda Expressions

Consider:

plus 1 1 = ($\underline{\text{Y}}$ $\underline{\text{rplus}}$) 1 1
→ $\underline{\text{rplus}}$ $\underline{\text{plus}}$ $\underline{\text{1}}$ $\underline{\text{1}}$
→ $\underline{\text{iszero}}$ $\underline{\text{1}}$
1
(plus (pred 1) (succ 1))
→ $\underline{\text{False}}$ $\underline{\text{1}}$ $\underline{\text{(plus (pred 1) (succ 1))}}$
→ $\underline{\text{plus}}$ (pred 1) (succ 1)
→ $\underline{\text{rplus}}$ $\underline{\text{plus}}$ $\underline{\text{(pred 1)}}$ $\underline{\text{(succ 1)}}$
→ iszero ($\underline{\text{pred 1}}$)
(succ 1)
(plus (pred (pred 1)) (succ (succ 1)))
→ $\underline{\text{iszero 0}}$
(succ 1)
(...)
→ $\underline{\text{True}}$ (succ 1) (...)
→ $\underline{\text{succ}}$ $\underline{\text{1}}$
→ 2

The Typed Lambda Calculus

There are many variants of the lambda calculus.
The *typed lambda calculus* decorates terms with type annotations:

Syntax:

$$e ::= x^{\tau} \mid e_1^{\tau 2 \rightarrow \tau 1}\ e_2^{\tau 2} \mid (\lambda\, x^{\tau 2}.e^{\tau 1})^{\tau 2 \rightarrow \tau 1}$$

Operational Semantics:

α conversion : $\lambda\, x^{\tau 2} . e^{\tau 1} \Leftrightarrow \lambda\, y^{\tau 2} . [\, y^{\tau 2}/x^{\tau 2}\,]\, e^{\tau 1}$ where $y^{\tau 2}$ is not free in $e^{\tau 1}$

β reduction: $(\lambda\, x^{\tau 2} . e_1^{\tau 1})\ e_2^{\tau 2} \Rightarrow [\, e_2^{\tau 2}/x^{\tau 2}\,]\, e_1^{\tau 1}$

η reduction: $\lambda\, x^{\tau 2}. (e^{\tau 1}\ x^{\tau 2}) \Rightarrow e^{\tau 1}$ if $x^{\tau 2}$ is not free in $e^{\tau 1}$

Example:

$$\text{True} \equiv (\, \lambda\, x^{A} . (\, \lambda\, y^{B} . x^{A}\,)^{B \rightarrow A})^{A \rightarrow (B \rightarrow A)}$$

The Polymorphic Lambda Calculus

Polymorphic functions like “map” cannot be typed in the typed lambda calculus!

Need *type variables* to capture polymorphism:

β reduction (ii): $(\lambda\, x^{\nu} \,.\, e_1^{\tau 1})\; e_2^{\tau 2} \Rightarrow [\,\tau 2 \,/\, \nu\,]\,[\,e_2^{\tau 2}/x^{\nu}\,]\; e_1^{\tau 1}$

Example:

$$\begin{aligned} \text{True} \quad &\equiv \quad (\,\lambda\, x^{\alpha} \,.\, (\,\lambda\, y^{\beta} \,.\, x^{\alpha}\,)^{\beta\rightarrow\alpha})^{\alpha\rightarrow(\beta\rightarrow\alpha)} \\ \text{True}^{\alpha\rightarrow(\beta\rightarrow\alpha)}\; a^{A}\; b^{B} \quad &\rightarrow \quad (\,\lambda\, y^{\beta} \,.\, a^{A}\,)^{\beta\rightarrow A}\; b^{B} \\ &\rightarrow \quad a^{A} \end{aligned}$$

Hindley-Milner Polymorphism

Hindley-Milner polymorphism (i.e., that adopted by ML and Haskell) works by inferring the type annotations for a slightly restricted subcalculus: polymorphic functions.

If:

```
dlen l l' xs ys = (l xs) + (l' ys)
```

then

```
dlen length length “aaa” [1,2,3]
```

is ok, but if

```
dlen' l xs ys = (l xs) + (l ys),
```

then

```
dlen' length “aaa” [1,2,3]
```

is a type error since the argument `l` cannot be assigned a unique type!

Polymorphism and self application

Even the polymorphic lambda calculus is not powerful enough to express certain lambda terms.

Recall that both Ω and the Y combinator make use of “self application”:

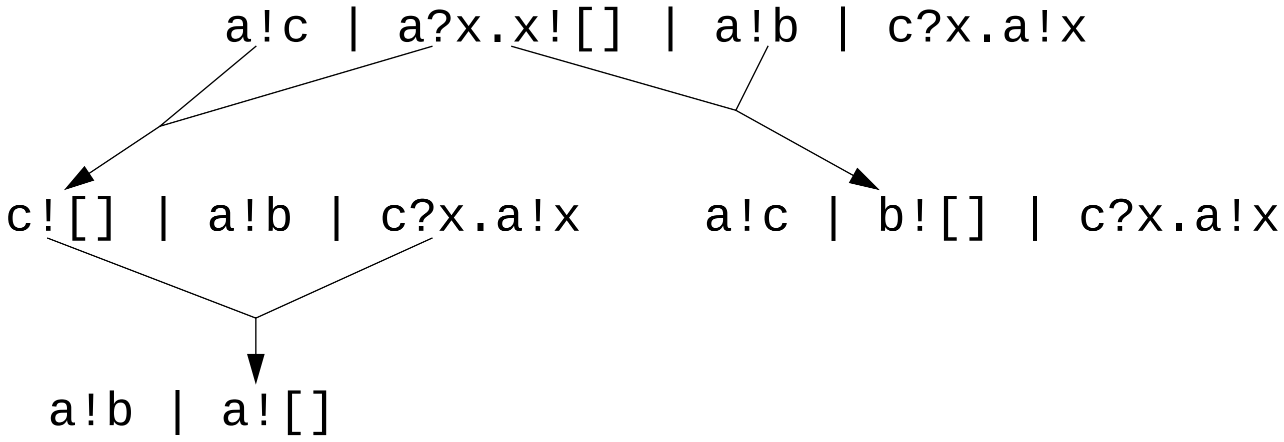
$$\Omega = (\lambda x . x x) (\lambda x . x x)$$

✎ *What type annotation would you assign to the expression* ($\lambda x . x x$)*?*

Process Calculi

Process calculi model *processes* rather than functions.

Since inter-process communication is inherently non-deterministic, the Church-Rosser property typically does not hold:



`a!c | a?x.x![] | a!b | c?x.a!x`

`c![] | a!b | c?x.a!x`

`a!c | b![] | c?x.a!x`

`a!b | a![]`

Process calculi are capable of modeling all computation in terms of communication.

Summary

You should know the answers to these questions:

- ❑ Why isn’t it possible to express recursion directly in the lambda calculus?
- ❑ What is a fixed point? Why is it important?
- ❑ How does the typed lambda calculus keep track of the types of terms?
- ❑ How does a polymorphic function differ from an ordinary one?

Can you answer the following questions?

- ✎ *Are there more fixed-point operators other than Y?*
- ✎ *How can you be sure that unfolding a recursive expression will terminate?*
- ✎ *How would you express the semantics of the example process calculus?*

8. Introduction to Denotational Semantics

Overview:

- Syntax and Semantics
- Approaches to Specifying Semantics
- Semantics of Expressions
- Semantics of Assignment
- Other Issues

References:

- D. A. Schmidt, *Denotational Semantics*, Wm. C. Brown Publ., 1986
- D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

Defining Programming Languages

Three main characteristics of programming languages:

1. **Syntax:** What is the appearance and structure of its programs?
2. **Semantics:** What is the *meaning* of programs?

   The *static semantics* tells us which (syntactically valid) programs are semantically valid (i.e., which are *type correct*) and the *dynamic semantics* tells us how to interpret the meaning of valid programs.
3. **Pragmatics:** What is the usability of the language?

   How easy is it to implement? What kinds of applications does it suit?

Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate to the implementors as well as to programmers.

A precise standard for a computer implementation:

- ☞ How should the language be implemented on different machines?

User documentation:

- ☞ What is the meaning of a program, given a particular combination of language features?

A tool for design and analysis:

- ☞ How can the language definition be tuned so that it can be implemented efficiently?

Input to a compiler generator:

- ☞ How can a reference implementation be obtained from the specification?

Methods for Specifying Semantics

Operational Semantics:

- ☞ ⟦ program ⟧ = *abstract machine program*
- ☞ can be simple to implement
- ☞ hard to reason about

Denotational Semantics:

- ☞ ⟦ program ⟧ = *mathematical denotation* (typically, a function)
- ☞ facilitates reasoning
- ☞ not always easy to find suitable semantic domains

Axiomatic Semantics:

- ☞ ⟦ program ⟧ = *set of properties*
- ☞ good for proving theorems about programs
- ☞ somewhat distant from implementation

Structured Operational Semantics:

- ☞ ⟦ program ⟧ = *transition system* (defined using inference rules)
- ☞ good for concurrency and non-determinism
- ☞ hard to reason about equivalence

Concrete and Abstract Syntax

How to parse “`4 * 2 + 1`”?

Abstract Syntax is compact but ambiguous:

Expr ::= Num
| Expr Op Expr
Op ::= +| - | * | /

Concrete Syntax is unambiguous but verbose:

Expr ::= Expr LowOp Term
| Term
Term ::= Term HighOp Factor
| Factor
Factor ::= Num
| (Expr)
LowOp ::= + | -
HighOp ::= * | /

Semantic Domains

In order to define semantic mappings of programs and their features to their mathematical denotations, the semantic domains must be precisely defined:

```
data Bool = True | False

(&&), (||) :: Bool -> Bool -> Bool
False &&    x      = False
True  &&    x      = x

False ||    x      = x
True  ||    x      = True


not :: Bool -> Bool
not   True         = False
not   False        = True
```

A Calculator Language

Abstract Syntax:

Prog ::= '`ON`' Stmt

Stmt ::= Expr '`TOTAL`' Stmt
| Expr '`TOTAL`' '`OFF`'

Expr ::= Expr_1 '`+`' Expr_2
| Expr_1 '`*`' Expr_2
| '`IF`' Expr_1 '`,`' Expr_2 '`,`' Expr_3
| '`LASTANSWER`'
| '`(`' Expr '`)`'
| Num

The program “ `ON 4 * ( 3 + 2 ) TOTAL OFF` ” should print out `20` and stop.

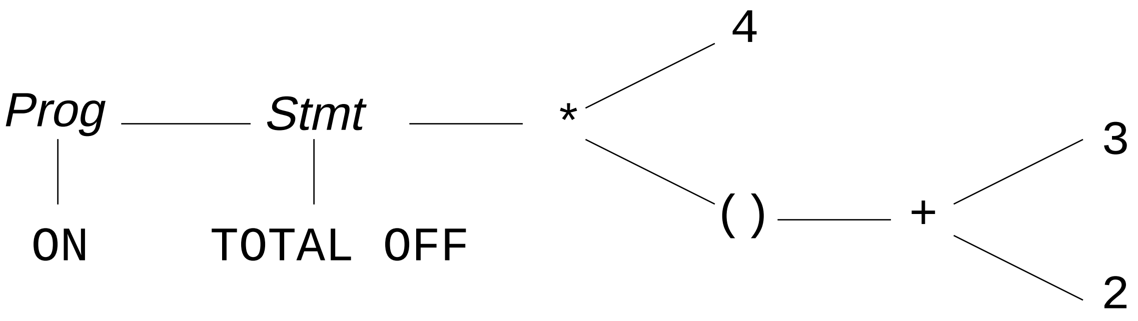
Data Structures for Syntax Tree

We can represent programs in our calculator language as syntax trees:

```
data Program      =  On ExprSequence
data ExprSequence =  Total Expression ExprSequence
                  |  TotalOff Expression
data Expression   =  Plus Expression Expression
                  |  Times Expression Expression
                  |  If Expression Expression Expression
                  |  LastAnswer
                  |  Braced Expression
                  |  N Int
```

Representing Syntax

The test program “ `ON 4 * ( 3 + 2 ) TOTAL OFF` ” can be parsed as:



And represented as:

```
test     =  On    (TotalOff  (Times   (N 4)
                             (Braced  (Plus (N 3)
                                            (N 2) ) ) ) )
```

Calculator Semantics

Programs:

$$
\begin{aligned}
&\mathbf{P} : \mathit{Program} \to \mathit{Int}\ ^{*} \\
&\mathbf{P}\,[\![\ \texttt{ON}\ S\ ]\!] && = \mathbf{S}\,[\![\ S\ ]\!]\ (0)
\end{aligned}
$$

Sequences:

$$
\begin{aligned}
&\mathbf{S} :: \mathit{ExprSequence} \to \mathit{Int} \to \mathit{Int}\ ^{*} \\
&\mathbf{S}\,[\![\ E\ \texttt{TOTAL}\ S\ ]\!]\ (n) && = \mathit{let}\ n' = \mathbf{E}\,[\![\ E\ ]\!]\ (n)\ \mathit{in}\ [\ n',\ \mathbf{S}\,[\![\ S\ ]\!]\ (n')\ ] \\
&\mathbf{S}\,[\![\ E\ \texttt{TOTAL}\ \texttt{OFF}\ ]\!]\ (n) && = [\ \mathbf{E}\,[\![\ E\ ]\!]\ (n)\ ]
\end{aligned}
$$

Expressions:

$$
\begin{aligned}
&\mathbf{E} : \mathit{Expression} \to \mathit{Int} \to \mathit{Int} \\
&\mathbf{E}\,[\![\ E1 + E2\ ]\!]\ (n) && = \mathbf{E}\,[\![\ E1\ ]\!]\ (n) + \mathbf{E}\,[\![\ E2\ ]\!]\ (n) \\
&\mathbf{E}\,[\![\ E1\ \texttt{*}\ E2\ ]\!]\ (n) && = \mathbf{E}\,[\![\ E1\ ]\!]\ (n) \times \mathbf{E}\,[\![\ E2\ ]\!]\ (n) \\
&\mathbf{E}\,[\![\ \texttt{IF}\ E1\ \texttt{,}\ E2\ \texttt{,}\ E3\ ]\!]\ (n) && = \mathbf{E}\,[\![\ E1\ ]\!]\ (n) = 0 \to \mathbf{E}\,[\![\ E2\ ]\!]\ (n)\ \#\ \mathbf{E}\,[\![\ E3\ ]\!]\ (n) \\
&\mathbf{E}\,[\![\ \texttt{LASTANSWER}\ ]\!]\ (n) && = n \\
&\mathbf{E}\,[\![\ \texttt{(}\ E\ \texttt{)}\ ]\!]\ (n) && = \mathbf{E}\,[\![\ E\ ]\!]\ (n) \\
&\mathbf{E}\,[\![\ N\ ]\!]\ (n) && = N
\end{aligned}
$$

Implementing the Calculator

Programs:

```
pp :: Program -> [Int]
pp (On s)               =     ss s 0
```

Sequences:

```
ss :: ExprSequence -> Int -> [Int]
ss (Total e s) n        =     let n' = (ee e n) in n' : (ss s n')
ss (TotalOff e) n       =     (ee e n) : [ ]
```

Expressions:

```
ee :: Expression -> Int -> Int
ee (Plus e1 e2) n       =     (ee e1 n) + (ee e2 n)
ee (Times e1 e2) n      =     (ee e1 n) * (ee e2 n)
ee (If e1 e2 e3) n
      | (ee e1 n) == 0 =     (ee e2 n)
      | otherwise       =     (ee e3 n)
ee (LastAnswer) n       =     n
ee (Braced e) n         =     (ee e n)
ee (N num) n            =     num
```

A Language with Assignment

Abstract Syntax:

Prog	::=	Cmd '`.`'
Cmd	::=	Cmd_1 '`;`' Cmd_2
	|	'`if`' Bool '`then`' Cmd_1 '`else`' Cmd_2
	|	Id '`:=`' Exp
Exp	::=	Exp_1 '+' Exp_2
	|	Id
	|	Num
Bool	::=	Exp_1 '=' Exp_2
	|	'`not`' Bool

Example:

“`z := 1 ; if a = 0 then z := 3 else z := z + a .`”

Programs take a single number as input, which initializes the variable ‘a’. The output of a program is the final value of the variable ‘z’.

Abstract Syntax Trees

Data Structures:

```
data Program     =     Dot Command
data Command     =     CSeq Command Command
                 |     Assign Identifier Expression
                 |     If BooleanExpr Command Command
data Expression  =     Plus Expression Expression
                 |     Id Identifier
                 |     Num Int
data BooleanExpr =     Equal Expression Expression
                 |     Not BooleanExpr
type Identifier  =     Char
```

Example:

```
Dot   (CSeq (Assign 'z' (Num 1))
            (If (Equal (Id 'a') (Num 0))
                  (Assign 'z' (Num 3))
                  (Assign 'z' (Plus (Id 'z') (Id 'a')))
            )
      )
```

Modelling Environments

A store is a mapping from identifiers to values:

```
type Store = Identifier -> Int

newstore :: Store
newstore id              =     0

access :: Identifier -> Store -> Int
access id store          =     store id

update :: Identifier -> Int -> Store -> Store
update id val store      =     store'
                                  where store' id'
                                     | id' == id = val
                                     | otherwise = store id'
```

Semantics of Assignments

```
pp :: Program -> Int -> Int
pp (Dot c) n             = access 'z' (cc c (update 'a' n newstore))

cc :: Command -> Store -> Store
cc (CSeq c1 c2) s        = cc c2 (cc c1 s)
cc (Assign id e) s       = update id (ee e s) s
cc (If b c1 c2) s        = ifelse (bb b s) (cc c1 s) (cc c2 s)

ee :: Expression -> Store -> Int
ee (Plus e1 e2) s        = (ee e2 s) + (ee e1 s)
ee (Id id) s             = access id s
ee (Num n) s             = n

bb :: BooleanExpr -> Store -> Bool
bb (Equal e1 e2) s       = (ee e1 s) == (ee e2 s)
bb (Not b) s             = not (bb b s)
ifelse :: Bool -> a -> a -> a
ifelse True x y          = x
ifelse False x y         = y
```

Practical Issues

Modelling:

- ❑ Errors and non-termination:
  - ☞ need a special “error” value in semantic domains
- ❑ Branching:
  - ☞ semantic domains in which “continuations” model “the rest of the program” make it easy to transfer control
- ❑ Interactive input
- ❑ Dynamic typing
- ❑ ...

Theoretical Issues

What are the denotations of lambda abstractions?

- need Scott’s theory of semantic domains

What is the semantics of recursive functions?

- need least fixed point theory

How to model concurrency and non-determinism?

- abandon standard semantic domains
- use “interleaving semantics”
- “true concurrency” requires other models ...

Summary

You should know the answers to these questions:

- What is the difference between syntax and semantics?
- What is the difference between abstract and concrete syntax?
- What is a semantic domain?
- How can you specify semantics as mappings from syntax to behaviour?
- How can assignments and updates be modelled with (pure) functions?

Can you answer the following questions?

- *Why are semantic functions typically higher-order?*
- *Does the calculator semantics specify strict or lazy evaluation?*
- *Does the implementation of the calculator semantics use strict or lazy evaluation?*
- *Why do commands and expressions have different semantic domains?*

9. Logic Programming

Overview

- ❑ Facts and Rules
- ❑ Resolution and Unification
- ❑ Searching and Backtracking
- ❑ Recursion, Functions and Arithmetic
- ❑ Lists and other Structures

References:

- ❑ Kenneth C. Louden, *Programming Languages: Principles and Practice*, PWS Publishing (Boston), 1993.
- ❑ Sterling and Shapiro, *The Art of Prolog*, MIT Press, 1986
- ❑ Clocksin and Mellish, *Programming in Prolog*, Springer Verlag, 1981

Logic Programming Languages

What is a Program?

A program is a database of facts (axioms) together with a set of inference rules for proving theorems from the axioms.

Imperative Programming:

☞ Program = Algorithms + Data

Logic Programming:

☞ Program = Facts + Rules

or

☞ Algorithms = Logic + Control

Prolog

A Prolog program consists of *facts*, *rules*, and *questions:*

- ❑ *Facts* are named *relations* between objects:

```
parent(charles, elizabeth).
% elizabeth is a parent of charles

female(elizabeth).
% elizabeth is female
```

- ❑ *Rules* are relations (goals) that can be inferred from other relations (subgoals):

```
mother(X, M) :- parent(X,M), female(M).
% M is a mother of X if M is a parent of X and M is female
```

- ❑ *Questions* are statements that can be answered using facts and rules:

```
?- parent(charles, elizabeth).
➪ yes


?- mother(charles, M).
➪ M = elizabeth
  yes
```

Horn Clauses

Both rules and facts are instances of *Horn clauses,* of the form:

A_0 if A_1 and A_2 and ... A_n

A_0 is the *head* of the Horn clause and “A_1 and A_2 and ... A_n” is the *body*

Facts are just Horn clauses with no body:

parent(charles, elizabeth)	if	True
female(elizabeth)	if	True
mother(X, M)	if	parent(X,M)
	and	female(M)

Resolution and Unification

Questions (or *goals*) are answered by *matching* goals against facts or rules, *unifying* variables with terms, and *backtracking* when subgoals fail.

If a subgoal of a Horn clause matches the head of another Horn clause, *resolution* allows us to replace that subgoal by the body of the matching Horn clause.

Unification lets us bind variables to corresponding values in the matching Horn clause:

	mother(charles, M)
	parent(charles, M) and female(M)
{ M = elizabeth }	True and female(elizabeth)
{ M = elizabeth }	True and True

Prolog Databases

A Prolog database is a file of facts and rules to be “consulted” before asking questions:

```
female(anne).
female(diana).
female(elizabeth).
male(andrew).
male(charles).
male(edward).
male(harry).
male(philip).
male(william).

parent(andrew, elizabeth).
parent(andrew, philip).
parent(anne, elizabeth).
parent(anne, philip).
parent(charles, elizabeth).
parent(charles, philip).
parent(edward, elizabeth).
parent(edward, philip).
parent(harry, charles).
parent(harry, diana).
parent(william, charles).
parent(william, diana).
```

```
?- consult('royal').
➪ yes
?- male(charles).
➪ yes
?- male(anne).
➪ no
?- male(mickey).
➪ no
?- parent(charles, P).
➪ P = elizabeth <carriage return>
  yes
?- male(X).
➪ X = andrew ;
  X = charles <carriage return>
  yes
?- parent(william, _).
➪ yes
```

Unification

Unification is the process of instantiating variables by pattern matching.

1. A constant unifies only with itself:

```
?- charles = charles.
➪ yes

?- charles = andrew.
➪ no
```

2. An uninstantiated variable unifies with anything:

```
?- parent(charles, elizabeth) = Y.
➪ Y = parent(charles,elizabeth) ?
  yes
```

3. A structured term unifies with another term only if it has the same function name and number of arguments, and the arguments can be unified recursively:

```
?- parent(charles, P) = parent(X, elizabeth).
➪ P = elizabeth,
  X = charles ?
  yes
```

Evaluation Order

In principle, any of the parameters in a query may be instantiated or not

```
?- mother(X, elizabeth).
➪ X = andrew ? ;
  X = anne ? ;
  X = charles ? ;
  X = edward ? ;
  no
```

```
?- mother(X, M).
➪ M = elizabeth,
  X = andrew ?
  yes
```

Prolog adopts a *closed world assumption* — whatever cannot be proved to be true, is assumed to be false.

```
?- mother(elizabeth,M).
➪ no
```

Backtracking

Prolog applies resolution in linear fashion, replacing goals left to right, and considering database clauses top-to-bottom.

```
father(X, M) :- parent(X,M), male(M).

?- trace.

➪ {The debugger will first creep -- showing everything (trace)}
  yes
  {trace}

?- father(charles,F).
➪ + 1 1 Call: father(charles,_67) ?
  + 2 2 Call: parent(charles,_67) ?
  + 2 2 Exit: parent(charles,elizabeth) ?
  + 3 2 Call: male(elizabeth) ?
  + 3 2 Fail: male(elizabeth) ?
  + 2 2 Redo: parent(charles,elizabeth) ?
  + 2 2 Exit: parent(charles,philip) ?
  + 3 2 Call: male(philip) ?
  + 3 2 Exit: male(philip) ?
  + 1 1 Exit: father(charles,philip) ?
F = philip ?
yes
{trace}
```

Comparison

The predicate = attempts to unify its two arguments:

```
?- X = charles.
➪ X = charles ?
  yes
```

The predicate == tests if the terms instantiating its arguments are literally identical:

```
?- charles == charles.
➪ yes


?- X == charles.
➪ no


?- X = charles, male(charles) == male(X).
➪ X = charles ?
  yes
```

The predicate \== tests if its arguments are *not* literally identical:

```
?- X = male(charles), Y = charles, X \== male(Y).
➪ no
```

Sharing Subgoals

Common subgoals can easily be factored out as relations:

```
sibling(X, Y) :-    mother(X, M), mother(Y, M),
                    father(X, F), father(Y, F),
                    X \== Y.


brother(X, B) :-    sibling(X,B), male(B).
uncle(X, U) :-      parent(X, P), brother(P, U).


sister(X, S) :-     sibling(X,S), female(S).
aunt(X, A) :-       parent(X, P), sister(P, A).
```

Disjunctions

One may define multiple rules for the same predicate, just as with facts:

```
isparent(C, P) :-                 mother(C, P).
isparent(C, P) :-                 father(C, P).
```

Disjunctions can also be expressed using the “;” operator:

```
isparent(C, P) :-                 mother(C, P); father(C, P).
```

Note that same information can be represented in various forms — we could have decided to express mother/2 and father/2 as facts, and parent/2 as a rule. Ask:

- Which way is it easier to express and maintain facts?
- Which way makes it faster to evaluate queries?

Recursion

Recursive relations are defined in the obvious way:

```
ancestor(X, A) :-   parent(X, A).
ancestor(X, A) :-   parent(X, P), ancestor(P, A).
```

```
?- ancestor(X, philip).
➪ + 1 1 Call: ancestor(_61,philip) ?
  + 2 2 Call: parent(_61,philip) ?
  + 2 2 Exit: parent(andrew,philip) ?
  + 1 1 Exit: ancestor(andrew,philip) ?
X = andrew ?
yes
```

```
?- ancestor(harry, philip).
➪ + 1 1 Call: ancestor(harry,philip) ?
  + 2 2 Call: parent(harry,philip) ?
  + 2 2 Fail: parent(harry,philip) ?
  + 2 2 Call: parent(harry,_316) ?
  + 2 2 Exit: parent(harry,charles) ?
  + 3 2 Call: ancestor(charles,philip) ?
  + 4 3 Call: parent(charles,philip) ?
  + 4 3 Exit: parent(charles,philip) ?
  + 3 2 Exit: ancestor(charles,philip) ?
  + 1 1 Exit: ancestor(harry,philip) ?
yes
```

Evaluation Order

Evaluation of recursive queries is sensitive to the order of the rules in the database, and when the recursive call is made:

```
anc2(X, A) :-    anc2(P, A), parent(X, P).
anc2(X, A) :-    parent(X, A).
```

```
?- anc2(harry, X).
➪ + 1 1 Call: anc2(harry,_67) ?
  + 2 2 Call: anc2(_325,_67) ?
  + 3 3 Call: anc2(_525,_67) ?
  + 4 4 Call: anc2(_725,_67) ?
  + 5 5 Call: anc2(_925,_67) ?
  + 6 6 Call: anc2(_1125,_67) ?
  + 7 7 Call: anc2(_1325,_67) ? abort
{Execution aborted}
```

Negation as Failure

Searching can be controlled by explicit failure:

```
printall(X) :-   X, print(X), nl, fail.
printall(_).
```

```
?- printall(brother(_,_)).
```

The *cut* operator (!) *commits* Prolog to a particular search path:

```
parent(C,P) :-   mother(C,P), !.
parent(C,P) :-   father(C,P).
```

Negation can be implemented by a combination of cut and fail:

```
not(X) :-        X, !, fail.
not(_).
```

Changing the Database

The Prolog database can be modified dynamically by means of assert and retract:

```
rename(X,Y) :- retract(male(X)), assert(male(Y)), rename(X,Y).
rename(X,Y) :- retract(female(X)), assert(female(Y)), rename(X,Y).
rename(X,Y) :- retract(parent(X,P)), assert(parent(Y,P)), rename(X,Y).
rename(X,Y) :- retract(parent(C,X)), assert(parent(C,Y)), rename(X,Y).
rename(_,_).
```

```
?- male(charles); parent(charles, _); parent(_, charles).
➪ yes
?- rename(charles, mickey).
➪ yes
?- male(charles); parent(charles, _); parent(_, charles).
➪ no
```

NB: With SICSTUS Prolog, such predicates must be declared dynamic:

```
:- dynamic male/1, female/1, parent/2.
```

Functions and Arithmetic

Functions are relations between expressions and values:

```
X is 5 + 6 .
```

Yields:

```
X = 11 ?
```

And is syntactic sugar for:

```
is(X, +(5,6))
```

User-defined functions are written in a relational style:

```
fact(0,1).
fact(N,F) :-     N > 0,
                 N1 is N - 1,
                 fact(N1,F1),
                 F is N * F1.
```

Lists

Lists are pairs of elements and lists:

Formal object	*Cons pair syntax*	*Element syntax*
.(a , [ ])	[a | [ ]]	[a]
.(a , .(b, [ ]))	[a | [b | [ ]]]	[a , b]
.(a , .(.(b , [ ]) , .(c , [ ])))	[a | [[b | [ ]] | [c | [ ]]]]	[a , [b] , c]
.(a , X)	[a | X]	[a | X]
.(a , .(b , X))	[a | [b | X]]	[a , b | X]

Pattern Matching with Lists

```
in(X, [X | _ ]).
in(X, [ _ | L]) :-     in(X, L).
```

```
?- in(b, [a,b,c]).
yes
```

```
?- in(X, [a,b,c]).
X = a ? ;
X = b ? ;
X = c ? ;
no
```

```
?- in(a, L).

L = [ a | _A ] ? ;
L = [ _A , a | _B ] ? ;
L = [ _A , _B , a | _C ] ? ;
L = [ _A , _B , _C , a | _D ] ?

yes
```

Exhaustive Searching

Searching for permutations:

```
perm([ ],[ ]).
perm([C|S1],S2) :-             perm(S1,P1),
                               append(X,Y,P1),
                               append(X,[C|Y],S2).

append([ ],L,L).
append([X|L1],L2,[X|L3]) :-  append(L1,L2,L3) .


?- printall(perm([a,b,c,d],_)).
```

A declarative, but hopelessly inefficient sort program:

```
ndsort(L,S) :-                 perm(L,S),
                               issorted(S).

issorted([ ]).
issorted([ _ ]).
issorted([N,M|S]) :-           N =< M,
                               issorted([M|S]).
```

Summary

You should know the answers to these questions:

- What are Horn clauses?
- What are resolution and unification?
- How does Prolog attempt to answer a query using facts and rules?
- When does Prolog assume that the answer to a query is false?
- When does Prolog backtrack? How does backtracking work?
- How are conjunction and disjunction represented?
- What is meant by “negation as failure”?
- How can you dynamically change the database?

Can you answer the following questions?

- *How can we view functions as relations?*
- *Is it possible to implement negation without either cut or fail?*
- *What happens if you use a predicate with the wrong number of arguments?*
- *What does Prolog reply when you ask* `not(male(X)).` *? What does this mean?*

10. Applications of Logic Programming

Overview

- ❑ I. Solving a puzzle:
    - ☞ SEND + MORE = MONEY
- ❑ II. Reasoning about functional dependencies:
    - ☞ finding closures, candidate keys and BCNF decompositions

References:

- ❑ A. Silberschatz, H.F. Korth and S. Sudarshan, *Database System Concepts*, 3d edition, McGraw Hill, 1997.

I. Solving a puzzle

- ✎ *Find values for the letters so the following equation holds:*

```
 SEND
+MORE
-----
MONEY
```

A non-solution:

We would *like* to write:

```
soln0 :-             A is 1000*S + 100*E + 10*N + D,
                     B is 1000*M + 100*O + 10*R + E,
                     C is 10000*M + 1000*O + 100*N + 10*E + Y,
                     C is A+B,
                     showAnswer(A,B,C).

showAnswer(A,B,C)    :- writeln([A, ‘ + ‘, B, ‘ = ‘, C]).

writeln([])          :- nl.
writeln([X|L])       :- write(X), writeln(L).
```

But this doesn’t work because “is” can only evaluate expressions over instantiated variables.

```
soln0.
➪ » evaluation_error: [goal(_1007 is 1000 * _1008 + 100 * _1009 + 10 * _1010 +
_1011),
   argument_index(2)]

   [Execution aborted]
```

A first solution

So let's instantiate them:

```
digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).

digits([]).    % everything in the argument list is a digit
digits([D|L]) :- digit(D), digits(L).

soln1 :-      digits([S,E,N,D,M,O,R,E,M,O,N,E,Y]), % pick arbitrary values
              A is 1000*S + 100*E + 10*N + D,
              B is 1000*M + 100*O + 10*R + E,
              C is 10000*M + 1000*O + 100*N + 10*E + Y,
              C is A+B,                            % check if solution is found
              showAnswer(A,B,C).
```

This is now correct, but yields a trivial solution!

```
soln1.
➪     0 + 0 = 0
      yes
```

A second (non-)solution

So let’s constrain S and M:

```
soln2 :-    digits([S,M]),
            not(S==0), not(M==0),              % backtrack if 0
            digits([N,D,M,O,R,E,M,O,N,E,Y]),
            A is 1000*S + 100*E + 10*N + D,
            B is 1000*M + 100*O + 10*R + E,
            C is 10000*M + 1000*O + 100*N + 10*E + Y,
            C is A+B,
            showAnswer(A,B,C).
```

Maybe it works. We’ll never know ...

```
soln2.
➪     [Execution aborted]
```

after 8 minutes still running ...

A third solution

Let’s try to exercise more control by instantiating variables bottom-up:

```
sum([],0).
sum([N|L], TOTAL) :-         % TOTAL is sum of numbers in [N|L]
            sum(L,SUBTOTAL), TOTAL is N + SUBTOTAL.

carrysum(L,D,C) :-           % Find D and C, where L is D + 10*C, digit(D)
            sum(L,S), C is S/10, D is S - 10*C.

soln3 :-    digits([D,E]), carrysum([D,E],Y,C1),
            digits([N,R]), carrysum([C1,N,R],E,C2),
            digit(O), carrysum([C2,E,O],N,C3),
            digits([S,M]), not(S==0), not(M==0),
            carrysum([C3,S,M],O,M),
            A is 1000*S + 100*E + 10*N + D,
            B is 1000*M + 100*O + 10*R + E,
            C is A+B,        % NB: we have dropped the evaluation of MONEY
            showAnswer(A,B,C).
```

This is also correct, but uninteresting:

```
soln3.
➪     9000 + 1000 = 10000
      yes
```

A fourth solution

Let's try to make the variables unique:

```
unique([]).             % There are no duplicate elements in the argument list
unique([X|L]) :- not(in(X,L)), unique(L).

in(X, [X|_]).           % X is in the argument list
in(X, [_|L]) :- in(X, L).

soln4 :-    L1 = [D,E], digits(L1), unique(L1),
            carrysum([D,E],Y,C1),
            L2 = [N,R,Y|L1], digits([N,R]), unique(L2),
            carrysum([C1,N,R],E,C2),
            L3 = [O|L2], digit(O), unique(L3),
            carrysum([C2,E,O],N,C3),
            L4 = [S,M|L3], digits([S,M]), not(S==0), not(M==0),
            unique(L4),
            carrysum([C3,S,M],O,M),
            A is 1000*S + 100*E + 10*N + D,
            B is 1000*M + 100*O + 10*R + E,
            C is A+B,
            showAnswer(A,B,C).
```

This works, in about 8 seconds on a PowerMac 7300/200:

```
soln4.
➪     9567 + 1085 = 10652
      yes
```

II. Reasoning about functional dependencies

We would like to represent functional dependencies for relational databases as Prolog terms, and write predicates that compute (i) closures of attribute sets, (ii) candidate keys and (iii) BCNF decompositions.

First, we would like to overload Prolog syntax as follows:

```
FDS = [ [a]->[b,c], [c,g]->[h,i], [b,c]->[h] ].
```

➪ *Syntax Error - unable to parse this character* » `->[b,c] ...`

but the built-in arrow operator has precedence higher than that of “,” and “=”:

```
op(1050, xfy, [ -> ]).
op(1000, xfy, [ ’,’ ]).
op(700, xfx, [ = ]).
```

so let’s change it:

```
% redefine precedence so -> has lower precedence than = or ,
:- op(600, xfx, [ -> ]).
```

Now we can get started ...

Computing closures

We would like to define a predicate:

`closure(FDS, AS, CS)`

which computes the closure CS of an attribute set AS using the dependencies FDS.

To do this, we should use Amstrong's axioms:

1. $B \subseteq A \quad \Rightarrow \quad A \to B$ (reflexivity)
2. $A \to B \quad \Rightarrow \quad AC \to BC$ (augmentation)
3. $A \to B,\ B \to C \quad \Rightarrow \quad A \to C$ (transitivity)

Intuitively, we add attributes to a set AS', using the axioms and the FDs, until no more dependencies can be applied:

- start with AS→AS', where AS' = AS (1)
- find some B→C, AS' = BD ⇒ AS→AS'→CD (2,3)
- repeat till no more FD applies

NB: each FD can be applied at most once!

A closure predicate

Try to express the algorithm declaratively:

```
closure(FDS, AS, CS) :-
   applies(FDS, B->C, AS, FDRest),    % Find some B->C in FDS that applies to AS
   union(AS, C, AS1),                 % Use it to augment AS to AS1
   closure(FDRest, AS1, CS).          % and continue with the remaining FDs
closure(FDS, AS, AS).                 % Else no FD applies, so we are done.


applies([B->C|FDS], B->C, AS, FDS) :-% FD applies to AS if B is a subset of AS
   subset(B,AS).
applies([FD|FDS], B->C, AS, [FD|FDRest]) :-
   applies(FDS, B->C, AS, FDRest).    % If first doesn’t apply, keep searching
```

Now we must worry about the details ...

Manipulating sets

We need some predicates to manipulate attribute sets and sets of FDs:

```
in(X, [X|_]).                 % in(X,S) -- X is in the argument list
in(X, [_|S]) :- in(X, S).


subset([],_).                 % subset(S1,S2) -- S1 is a subset of S2
subset([X|S1],S2) :-
   in(X,S2),
   subset(S1,S2).


rem(_,[],[]) .                % rem(X,S,R) -- removing X from S yields R
rem(X,[X|S],R) :- rem(X,S,R), !.
rem(X,[Y|S],[Y|R]) :- rem(X,S,R) .


union([],S,S) .               % union(S1,S2,U) -- U is the union of S1 and S2
union([X|S1],S2,U) :-
   rem(X,S2,S),               % transfer elements of S1 to S2 till S1 is empty
   union(S1,[X|S],U).
```

✎ *How would you express set difference and intersection?*

Evaluating closures

A couple of test cases:

```
showclosure(FDS, AS) :-
   closure(FDS, AS, CS),
   writeln([AS -> CS]).        % calls write() for each element, then nl.

find1 :-
   FDS = [  [a]->[b,c],
            [c,g]->[h,i],
            [b,c]->[h]    ],
   writeln([‘FDS = ‘, FDS]),
   showclosure(FDS, [a]),
   showclosure(FDS, [a,c]),
   showclosure(FDS, [a,g]).
```

find1.

```
➪     FDS = [[a]->[b,c],[c,g]->[h,i],[b,c]->[h]]
      [a]->[c,b,a,h]
      [a,c]->[b,a,c,h]
      [a,g]->[i,h,g,a,b,c]
```

Finding keys

Now we would like a predicate `candkey/2` that suggests a candidate key for the attributes in a set of FDs:

```
candkey(FDS, Key) :-
   attset(FDS, AS),                  % Find the set of all attributes in FDS
   minkey(FDS, AS, AS, Key).         % Find a minimal key, starting with AS

minkey(FDS, AS, Key, MinKey) :-      % Key is some key for AS; MinKey is minimal
   smallerkey(FDS, AS, Key, SmallerKey), !,   % Is there a smaller key?
   minkey(FDS, AS, SmallerKey, MinKey).       % if so, then try again
minkey(FDS, AS, MinKey, MinKey).     % else we are done!

smallerkey(FDS, AS, Key, Smaller) :-
   in(X, Key), rem(X, Key, Smaller), % Remove some X from Key
   iskey(Smaller, AS, FDS).          % Do we still have a key for AS?
iskey(Key, AS, FDS) :-               % Key is a key for att set AS wrt FDS
   closure(FDS, Key, Closure),       % The closure of Key must contain AS
   subset(AS, Closure).
```

✎ *How would you implement* `attset/2`*?*

Evaluating candidate keys

Two examples:

```
find2 :-
   FDS = [ [a]->[b,c], [c,g]->[h,i], [b,c]->[h] ], writeln([‘FDS = ‘, FDS]),
   candkey(FDS, Key), writeln([‘Key = ‘, Key]).
```

find2.

```
➪     FDS = [[a]->[b,c],[c,g]->[h,i],[b,c]->[h]]
      Key = [a,g]
```

```
find3 :-
   FDS = [ [a,b]->[c], [b]->[d], [e]->[f], [c,e]->[a] ], writeln([‘FDS = ‘, FDS]),
   candkey(FDS, Key), writeln([‘Key = ‘, Key]).
```

find3.

```
➪     FDS = [[a,b]->[c],[b]->[d],[e]->[f],[c,e]->[a]]
      Key = [a,b,e]
```

Testing for BCNF

Recall that a relation scheme is in BCNF if all non-trivial FDs define keys:

```
isbcnf(FDS, RS) :-
   fdsok(FDS, FDS, RS), !,          % RS is BCNF if all FDS are OK
   writeln([RS, ‘ is in BCNF’]).
isbcnf(FDS, RS) :-
   writeln([RS, ‘ is NOT in BCNF’]).


fdsok([], _, RS).                   % Nothing to check, so must be OK
fdsok([A->B|ToCheck], FDS, RS) :-
   subset(B,A),                     % A->B is trivial, so continue
   fdsok(ToCheck,FDS,RS).
fdsok([A->B|ToCheck], FDS, RS) :-   % Else check if A is a key
   iskey(A, RS, FDS),               % A is a key for RS, so OK
   fdsok(ToCheck,FDS,RS).           % check the others
```

Evaluating the BCNF test

An example from the database course:

```
check1 :-
   FDS = [  [branchName] -> [assets, branchCity],
            [loanNumber] -> [amount, branchName],
            [customerName] -> [customerName]        ],
   BranchScheme = [ branchName, assets, branchCity ],
   isbcnf(FDS, BranchScheme),
   BorrowScheme = [branchName, loanNumber, customerName, amount],
   isbcnf(FDS, BorrowScheme).
```

`check1.`

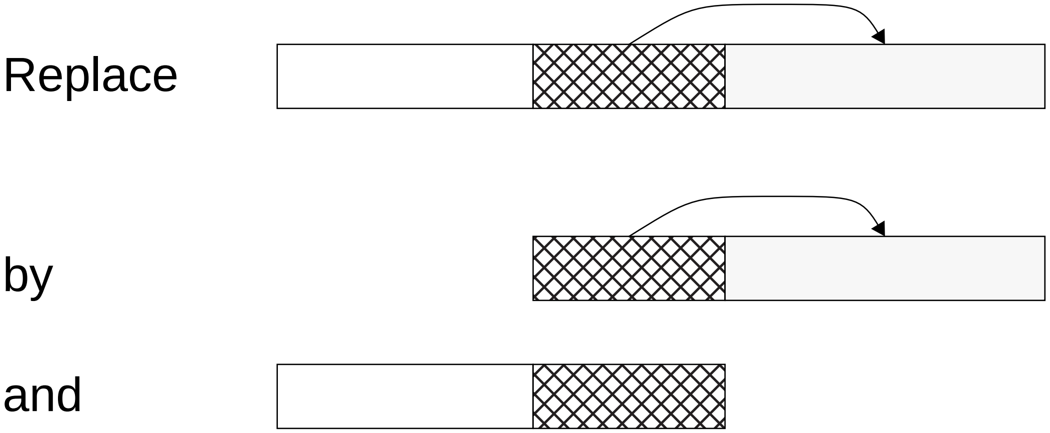
```
➪    [branchName,assets,branchCity] is in BCNF
     [branchName,loanNumber,customerName,amount] is NOT in BCNF
```

✎ *What would you modify to have* `isbcnf/2` *report exactly* <u>*which*</u> *FD is problematic?*

BCNF decomposition

Recall that BCNF decomposition works as follows:

```
while some R is not in BCNF
        select non-trivial $\alpha \rightarrow \beta$ holding on R where
                $\alpha \rightarrow R$ is not in $F^+$ and $\alpha \cap \beta = \emptyset$
        replace R by $\alpha \cup \beta$ and (R-$\beta$)
```



Replace

by

and

The trick is that $\alpha \rightarrow \beta$ may not be explicitly in the list F of FDs, and it is too expensive to compute the closure F^+

BCNF decomposition predicate

To decompose a schema, we must iterate through *both* the FDS *and* the schema.

```
bcnf(FDS, Decomp) :-                     % Decomp is the decomposition of FDS
   attset(FDS, AS),                      % Assume all attributes are in FDS
   writeln([‘Attribute set is ‘, AS]),
   bcnfDecomp(FDS, [AS], Decomp).        % Start with the trivial decomposition
```

Iterate through the schemas:

```
bcnfDecomp(FDS, [], []).                 % Nothing to decompose

bcnfDecomp(FDS, [RS|Schema], Decomp) :-  % If RS is not BCNF, then decompose it
   findBad(A->B, FDS, FDS, RS),          % Find a “bad” FD in FDS
   union(A,B,AB),
   diff(RS,B,Diff),
   writeln([‘Use ‘, A->B, ‘ to split ‘, RS, ‘ into ‘, AB, ‘ and ‘, Diff]), nl,
   bcnfDecomp(FDS,[AB,Diff|Schema],Decomp).% Decompose and start over

bcnfDecomp(FDS, [RS|Schema], [RS|Decomp]) :-% RS is in BCNF, so check rest
   bcnfDecomp(FDS, Schema, Decomp).
```

Finding “bad” FDs

For a given RS, we iterate through the FDs.

The “bad” FDs needed for decomposition may need to be *derived* from those we have.

```
findBad(A->B, [FD|FDS], AllFDS, RS) :-      % A->B is a “bad” FD
   FD = A->B0,                              % Try to derive a bad FD ...
   subset(A,RS),                            % A must apply to RS
   diff(B0,A,B1),                           % A ∩ B should be empty
   inter(B1,RS,B),                          % we are only interested in RS
   not(subset(B,A)),                        % A-> must not be trivial
   not(iskey(A, RS, AllFDS)).               % A->B is “bad” if A is not a key
                                            % for RS


findBad(FD, [OK|FDS], AllFDS, RS) :-        % First FD is OK, so check others
   findBad(FD, FDS, AllFDS, RS).
```

✎ *Can you justify the derivation of A->B using Armstrong’s axioms?*

Evaluating BCNF decomposition

The example from the database course:

```
check2 :-
   FDS = [  [branchName] -> [assets, branchCity],
            [loanNumber] -> [amount, branchName],
            [customerName] -> [customerName] ], % cheat to get this attribute in!
   bcnf(FDS, BCNF), writeln([‘BCNF decomposition of ‘, FDS, ‘ is ‘, BCNF]).
```

check2.

```
➪     Attribute set is [branchCity,assets,branchName,amount,loanNumber,customer-
Name]
      Use [branchName]->[assets,branchCity]
      to split [branchCity,assets,branchName,amount,loanNumber,customerName]
      into [branchName,assets,branchCity]
      and [branchName,amount,loanNumber,customerName]

      Use [loanNumber]->[amount,branchName]
      to split [branchName,amount,loanNumber,customerName]
      into [loanNumber,amount,branchName]
      and [loanNumber,customerName]

      BCNF decomposition of [[branchName]->[assets,branchCity],
         [loanNumber]->[amount,branchName], [customerName]->[customerName]]
      is [[branchName,assets,branchCity], [loanNumber,amount,branchName],
         [loanNumber,customerName]]
```

A final example

And finally, a more abstract, toy example:

```
check3 :-
   FDS = [  [a,b]->[c],
            [b]->[d],
            [e]->[f],
            [c,e]->[a]     ],
   bcnf(FDS, BCNF), writeln([‘BCNF decomposition of ‘, FDS, ‘ is ‘, BCNF]).
```

```
check3.
➪     Attribute set is [c,a,b,d,f,e]
      Use [a,b]->[c] to split [c,a,b,d,f,e] into [b,a,c] and [a,b,d,f,e]
      Use [b]->[d] to split [a,b,d,f,e] into [b,d] and [a,b,f,e]
      Use [e]->[f] to split [a,b,f,e] into [e,f] and [a,b,e]
      BCNF decomposition of [[a,b]->[c],[b]->[d],[e]->[f],[c,e]->[a]]
      is [[b,a,c],[b,d],[e,f],[a,b,e]]
```

✎ *What would you change in order to find all BCNF decompositions?*

Summary

Can you answer the following questions?

- ✎ *What happens when we ask* `digits([A,B,A])`*?*
- ✎ *How many times will* `soln2` *backtrack before finding a solution?*
- ✎ *How would you check if the solution to the puzzle is unique?*
- ✎ *How would you generalize the puzzle solution to solve arbitrary additions?*
- ✎ *The predicate* `in/2` *can be used both to check if an element is in a list, and to select elements from a list. Does* `subset/2` *also have this property? Why or why not?*
- ✎ *Can you justify that each of the recursive predicates will terminate?*
- ✎ *What would you do if you couldn't change the precedence of* `->/2`*?*
- ✎ *Can you verify that the* `closure/3` *predicate is correct?*
- ✎ *What would happen if we didn't cut in* `minkey/4`*?*
- ✎ *Would it be just as easy to implement these solutions with a functional language?*

11. Symbolic Interpretation

Overview

- ❑ Interpretation as Proof
- ❑ Operator precedence: representing programs as syntax trees
- ❑ An interpreter for the calculator language
- ❑ Implementing a Lambda Calculus interpreter
- ❑ Examples of lambda programs ...

Interpretation as Proof

One can view the execution of a program as a step-by-step “proof” that the program reaches some terminating state, while producing output along the way.

☞ The program and its intermediate states are represented as structures (typically, as syntax trees)

☞ Inference rules express how one program state can be transformed to the next

Representing Programs as Trees

Recall our Calculator example [Schmidt]:

```
P     ::=   'on' S
S     ::=   E 'total' S        |     E 'total' 'OFF'
E     ::=   E1 '+' E2          |     E1 '*' E2   |     'if' E1 'then' E2 'else' E3
      |     'lastanswer'       |     '(' E ')'   |     N
```

Syntax trees can be modelled directly as Prolog terms. For example, the program:

```
on 2+3 total lastanswer + 1 total off
```

can be modelled by the term:

```
on(total(2+3, total(lastanswer+1, off)))
```

Prefix and Infix Operators

Operator type and precedence can be defined to achieve convenient syntax:

```
:- op(900, fx,  on).
:- op(800, xfy, total).
:- op(600, fx,  if).
:- op(590, xfy, then).
:- op(580, xfy, else).

% op(500,  yfx, +).                           % these are pre-defined ...
% op(400,  yfx, *).
```

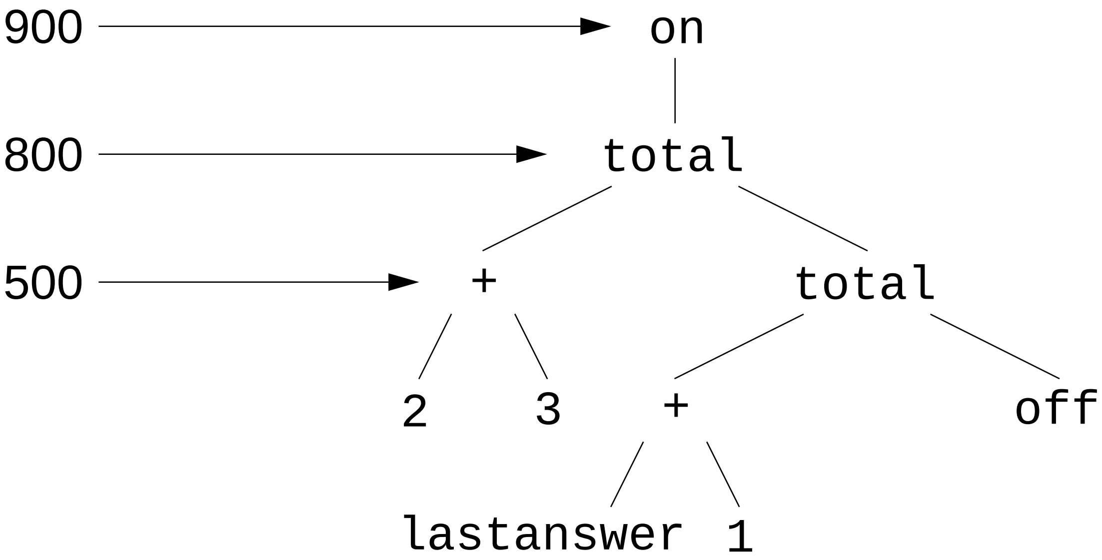
The higher the precedence, the higher in the syntax tree the operator will appear.

Operators can be declared (i) fx for prefix, (ii) xfy for right-associative, (iii) yfx for left-associative (iv) xfx for non-associating:

```
?- 1+2+3*4 = +(+(1,2),*(3,4)).
➪  yes


?- (on 2+3 total lastanswer+1 total off)
   == on(total(2+3, total(lastanswer+1, off))).
➪  yes
```

Operator precedence

```
on 2+3 total lastanswer+1 total off)
   == on(total(2+3, total(lastanswer+1, off))).
```



Standard Operators

The following operator precedences are predefined for SICSTUS Prolog:

```
op(1200, xfx,  [ :- , -- ]).
op(1200, fx,   [ :- , ?- ]).
op(1150, fx,   [ mode , public , dynamic , multifile , parallel , wait ]).
op(1100, xfy,  [ ; ]).
op(1050, xfy,  [ -> ]).
op(1000, xfy,  [ ’,’ ]).
op(900,  fy,   [ \+ , spy , nospy ]).
op(700,  xfx,  [ =, is, =.., ==, \==, @<, @>, @=<, @>=, =:=, =\=, <, >, =<, >= ]).
op(500,  yfx,  [ +, - , /\ , \/ ]).
op(500,  fx,   [ + , - ]).
op(400,  yfx,  [ * , / , // , << , >> ]).
op(300,  xfx,  [ mod ]).
op(200,  xfy,  [ ^ ]).
```

Building a Simple Interpreter

Top level programs:

```
on S :- seval(S, 0).
```

Statements:

```
seval(E total off, Prev) :-                 xeval(E, Prev, Val),
                                            print(Val), nl.
seval(E total S, Prev) :-                   xeval(E, Prev, Val),
                                            print(Val), nl,
                                            seval(S, Val).
```

Expressions:

```
xeval(N, _, N) :-                           number(N).
xeval(E1+E2, Prev, V) :-                    xeval(E1, Prev, V1),
                                            xeval(E2, Prev, V2),
                                            V is V1+V2.
xeval(E1*E2, Prev, V) :-                    xeval(E1, Prev, V1),
                                            xeval(E2, Prev, V2),
                                            V is V1*V2.
xeval(lastanswer, Prev, Prev).
xeval(if E1 then E2 else _, Prev, Val) :-   xeval(E1, Prev, 0),!,
                                            xeval(E2, Prev, Val).
xeval(if _ then _ else E3, Prev, Val) :-    xeval(E3, Prev, Val).
```

Running the Interpreter

```
?- on 2+3 total off.
   + 1 1 Call: on 2+3 total off ?
   + 2 2 Call: seval(2+3 total off,0) ?
   + 3 3 Call: xeval(2+3,0,_660) ?
   + 4 4 Call: number(2+3) ?
   + 4 4 Fail: number(2+3) ?
   + 4 4 Call: xeval(2,0,_892) ?
   + 5 5 Call: number(2) ?
   + 5 5 Exit: number(2) ?
   + 4 4 Exit: xeval(2,0,2) ?
   + 6 4 Call: xeval(3,0,_885) ?
   + 7 5 Call: number(3) ?
   + 7 5 Exit: number(3) ?
   + 6 4 Exit: xeval(3,0,3) ?
   + 8 4 Call: _660 is 2+3 ?
   + 8 4 Exit: 5 is 2+3 ?
   + 3 3 Exit: xeval(2+3,0,5) ?
   + 9 3 Call: print(5) ?
   + 9 3 Exit: print(5) ? 5
   + 10 3 Call: nl ?
   + 10 3 Exit: nl ?
   + 2 2 Exit: seval(2+3 total off,0) ?
   + 1 1 Exit: on 2+3 total off ?
yes
```

Lambda Calculus Interpreter

A somewhat more ambitious example is a Lambda Calculus interpreter.

First we must choose a syntax for lambda expressions:

```
:- op(600, fx, \).              % abstraction
:- op(650, xfy, :).             % body of abstraction
:- op(500, yfx, @).             % application
```

We cannot write `e1 e2` in Prolog, so we must introduce an operator for application.

For example, we will represent the lambda expression:

$$(\lambda x . \lambda y . x\ y)\ y$$

by the Prolog term:

```
(\x: \y: x@y) @ y      == @(:(\(x),:(\(y),@(x,y))), y).
```

Semantics

Alpha, beta and eta conversion are expressed as predicates over the “before” and “after” forms of lambda expressions:

```
alpha(\X:E, \Y:EY) :-     fv(E, FE),
                          not(in(Y, FE)),
                          subst(Y, X, E, EY).
beta((\X:E1)@E2, E3) :-   subst(E2, X, E1, E3).
eta(\X:E@X, E) :-         fv(E, F),
                          not(in(X, F)).
```

Free Variables

To implement conversion and reduction, we need to know the free variables in an expression:

```
fv(X, [X]) :-           isname(X).
fv(E1@E2, F12) :-       fv(E1, F1),
                        fv(E2, F2),
                        union(F1, F2, F12).
fv(\X:E, F) :-          isname(X),
                        fv(E, FE),
                        diff(FE, [X], F).
isname(N) :-            atom(N); number(N).
```

For example:

```
?- fv(\x: \y:x@y@z , F).
➪  F = [z] ?
   yes
```

Substitution

The predicate subst(E, X, EX, EE) is true if substituting E for X in EX yields EE:

```
subst(E, X, X, E) :-                 isname(X), !.
subst(E, X, Y, Y) :-                 isname(X), isname(Y),
                                     X \== Y.
subst(E, X, E1@E2, EE1@EE2) :-       subst(E, X, E1, EE1),
                                     subst(E, X, E2, EE2).
subst(E, X, \X:E1, \X:E1).
subst(E, X, \Y:E1, \Y:EE1) :-        X \== Y,
                                     fv(E, FE),
                                     not(in(Y, FE)), !,
                                     subst(E, X, E1, EE1).
```

This rule avoid name capture by substituting Y by a new name Z:

```
subst(E, X, \Y:E1, \Z:EEZ) :-        X \== Y,
                                     fv(E, FE),
                                     % in(Y, FE),
                                     fv(E1, F1),
                                     union(FE, F1, FU),
                                     newname(Y, Z, FU),
                                     subst(Z, Y, E1, EZ),
                                     subst(E, X, EZ, EEZ).
```

Renaming

newname(Y, Z, F) is true if Z is a new name for Y, not in F

```
newname(Y, Y, F) :-    not(in(Y, F)), !.
newname(Y, Z, F) :-    tick(Y, T), newname(T, Z, F).
```

The built-in predicate name(X, L) is true if the name X is represented by the ASCII list L

tick(Y, Z) is true if Z is Y with a “tick” (' = ASCII 39) appended

```
tick(Y, Z) :-          name(Y, LY), append(LY, [39], LZ), name(Z, LZ).
```

For example:

```
?- tick(x, Y).
➪  Y = x′ ?
   yes


?- subst(x@y, z, \x:x@z, E).
➪  E = \x':x'@(x@y)
   yes
```

Normal Form Reduction

E => NF is true if E reduces to normal form NF;
lazy(E, EE) is true if E reduces to EE by one normal-order reduction:

```
:- op(900, xfx, =>).

E => NF :-             lazy(E, EE), !, EE => NF.
X => X.                % no more reductions possible, so stop

lazy(E1, E2) :-        beta(E1, E2), !.
lazy(E1, E2) :-        eta(E1, E2), !.
lazy(E0@E2, E1@E2) :-  lazy(E0, E1), !.
```

For example:

```
?- (\x : (\y:x)@(\x:x)@x ) @ y => E.
➪  E = y@y ?
   yes
```

Viewing Intermediate States

The => predicate tells us what normal form a lambda expression reduces to, but does not tell us what reductions take us there.

To see intermediate reductions, we can print out each step:

```
:- op(800, fx, eval).

eval E :-              lazy(E, EE), !,
                       write(E), nl, write('-> '),
                       eval EE.

eval E :-              write(E), nl, write('STOP'), nl.
```

The same example yields:

```
?- eval (\x: \y: x@y) @ y.

➪  (\x: \y:x@y)@y
      -> \y':y@y'
      -> y
      STOP
```

Lazy Evaluation

The lambda expression $\Omega = (\lambda x . x\ x)\ (\lambda x . x\ x)$ has no normal form:

```
?- W = ((\x:x@x) @ (\x:x@x)),
   eval W.
➪  (\x:x@x)@(\x:x@x)
      -> (\x:x@x)@(\x:x@x)
      -> (\x:x@x)@(\x:x@x)
      <interrupt>
[Execution aborted]
```

But lazy evaluation allows it to be passed as a parameter if unused:

```
?- W = ((\x:x@x) @ (\x:x@x)),
   eval (\x:y) @ W.
➪  (\x:y)@((\x:x@x)@(\x:x@x))
      -> y
      STOP
```

Booleans

Recall the standard encoding of Booleans as lambda expressions that return their first (or second) argument:

```
?- True = \x: \y:x,
   False = \x: \y:y,
   Not = \b:b@False@True,
   eval Not@True.
```

```
➪  (\b:b@(\x: \y:y)@(\x: \y:x))@(\x: \y:x)
      -> (\x: \y:x)@(\x: \y:y)@(\x: \y:x)
      -> (\y: \x: \y:y)@(\x: \y:x)
      -> \x: \y:y
      STOP
```

Tuples

Recall that tuples can be modelled as higher-order functions that pass the values they hold to another (client) function:

```
?- True = \x: \y:x, False = \x: \y:y,
   Pair = (\x: \y: \z: z@x@y),
   First = (\p:p @ True),
   eval First @ (Pair @ 1 @ 2).

➪  (\p:p@(\x: \y:x))@((\x: \y: \z:z@x@y)@1@2)
      -> (\x: \y: \z:z@x@y)@1@2@(\x: \y:x)
      -> (\y: \z:z@1@y)@2@(\x: \y:x)
      -> (\z:z@1@2)@(\x: \y:x)
      -> (\x: \y:x)@1@2
      -> (\y:1)@2
      -> 1
      STOP
```

Natural Numbers

And natural numbers can be modelled using the standard encoding (though you probably won’t like what you see!):

```
?- True = \x: \y:x, False = \x: \y:y,
   Pair = (\x: \y: \z: z@x@y), First = (\p:p @ True), Second = (\p:p @ False),
   Zero = \x:x, Succ = \n:Pair@False@n, Succ@Zero => One,
   IsZero = First, Pred = Second,
   eval IsZero@(Pred@One).

➪  (\p:p@(\x: \y:x))@((\p:p@(\x: \y:y))@(\z:z@(\x: \y:y)@(\x:x)))
      -> (\p:p@(\x: \y:y))@(\z:z@(\x: \y:y)@(\x:x))@(\x: \y:x)
      -> (\z:z@(\x: \y:y)@(\x:x))@(\x: \y:y)@(\x: \y:x)
      -> (\x: \y:y)@(\x: \y:y)@(\x:x)@(\x: \y:x)
      -> (\y:y)@(\x:x)@(\x: \y:x)
      -> (\x:x)@(\x: \y:x)
      -> \x: \y:x
      STOP

yes
```

Fixed Points

Recall that we could not model the fixed point combinator Y in Haskell because self-application cannot be typed.

In our untyped interpreter, we can implement Y:

```
?- Y = \f:(\x:f@(x@x))@(\x:f@(x@x)),
   FP = Y@e,
   eval FP.

➪  (\f:(\x:f@(x@x))@(\x:f@(x@x)))@e
      -> (\x:e@(x@x))@(\x:e@(x@x))
      -> e@((\x:e@(x@x))@(\x:e@(x@x)))
      STOP
```

Note that this sequence validates that e@FP <-> FP.

Recursive Functions as Fixed Points ...

```
?- True = \x: \y:x, False = \x: \y:y,
   Pair = (\x: \y: \z: z@x@y), First = (\p:p @ True), Second = (\p:p @ False),
   Zero = \x:x, Succ = \n:Pair@False@n, Succ@Zero => One,
   IsZero = First, Pred = Second,
   Y = \f:(\x:f@(x@x))@(\x:f@(x@x)),
   RPlus = \plus: \n: \m : IsZero@n @m @(plus @ (Pred@n)@(Succ@m)),
   Y@RPlus => FPlus, FPlus@One@One => Two,
   eval IsZero@(Pred@(Pred@Two)).
```

```
➪  (\p:p@(\x: \y:x))@((\p:p@(\x: \y:y))
                 @((\p:p@(\x: \y:y))@(\z:z@(\x: \y:y)@(\z:z@(\x: \y:y)@(\x:x)))))
   -> (\p:p@(\x: \y:y))
      @ ((\p:p@(\x: \y:y))@(\z:z@(\x: \y:y)@(\z:z@(\x: \y:y)@ (\x:x)))) @ (\x: \y:x)
   -> (\p:p@(\x: \y:y)) @ (\z:z@(\x: \y:y)@(\z:z@(\x: \y:y)@(\x:x)))
         @ (\x: \y:y)@(\x: \y:x)
   -> (\z:z@(\x: \y:y)@(\z:z@(\x: \y:y)@(\x:x)))@(\x: \y:y)@(\x: \y:y)@(\x: \y:x)
   -> (\x: \y:y)@(\x: \y:y)@(\z:z@(\x: \y:y)@(\x:x))@(\x: \y:y)@(\x: \y:x)
   -> (\y:y)@(\z:z@(\x: \y:y)@(\x:x))@(\x: \y:y)@(\x: \y:x)
   -> (\z:z@(\x: \y:y)@(\x:x))@(\x: \y:y)@(\x: \y:x)
   -> (\x: \y:y)@(\x: \y:y)@(\x:x)@(\x: \y:x)
   -> (\y:y)@(\x:x)@(\x: \y:x)
   -> (\x:x)@(\x: \y:x)
   -> \x: \y:x
   STOP
```

Summary

You should know the answers to these questions:

- How can you represent programs as syntax trees? How can you represent syntax trees as Prolog terms?
- How can you define the syntax of your own language in Prolog?
- Why did we define ":" as right associate but "`@`" as left-associative?
- What is the difference between `Succ@Zero=>One` and `One=Succ@Zero`?

Can you answer the following questions?

- *How would you implement an interpreter for the assignment language we defined earlier?*
- *Why didn't we use "." in our syntax for lambda expressions?*
- *Does the order of the* `fv/2` *rules matter? What about* `subst/4`*?*
- *Can you explain each usage of "cut" (!) in the lambda interpreter?*
- *Can you think of other ways to implement* `newname/3`*?*
- *How would you modify the lambda interpreter to use strict evaluation?*

12. Scripting

Overview

- Scripting vs. Programming
- Python — an object-oriented scripting language
- Example: gluing web objects with Python

References:

- Guido van Rossum, *Python Tutorial*, Stichting Mathematisch Centrum, Amsterdam, 1996.
- Guido van Rossum, *Python Reference Manual*, Stichting Mathematisch Centrum, Amsterdam, 1996.
- Guido van Rossum, *Python Library Reference*, Stichting Mathematisch Centrum, Amsterdam, 1996.
- Aaron Watters , Guido van Rossum and James C. Ahlstrom, *Internet Programming with Python*, M&T Books, 1996.
- Mark Lutz, *Programming Python*, O'Reilly, 1996.

Scripting vs. Programming

Whereas a general-purpose programming language can be used to write standalone applications, the main purpose of a *scripting language* is to be “glue” components that are written in other language.

- Unix shell: glues Unix programs written in C or other languages
- TCL: glues C libraries, e.g. TK interface to X Window system
- Applescript: glues Macintosh applications
- Visual basic: glues COM, ActiveX components

A scripting language can often be used as an *embedding language*, allowing an application to be scriptable:

- Emacs editor: scriptable by EMACS Lisp
- Alpha editor: scriptable by TCL

The distinction is not always clear — e.g., Smalltalk is also used as a “glue language”, and Python and Perl can be used as general-purpose programming languages ...

Python

Python is an object-oriented scripting language that supports both scripting and programming-in-the-large:

Scripting features:

- ❑ Built-in high-level abstractions: strings, big numbers, lists and dictionaries
- ❑ Standard libraries: files, strings, regular expressions, math, time, threads, sockets, CGI, http, ftp, HTML parsing ...
- ❑ Compilation to byte-code, garbage collection
- ❑ Dynamically bound names, run-time type-checking, “eval”

Programming-in-the-large:

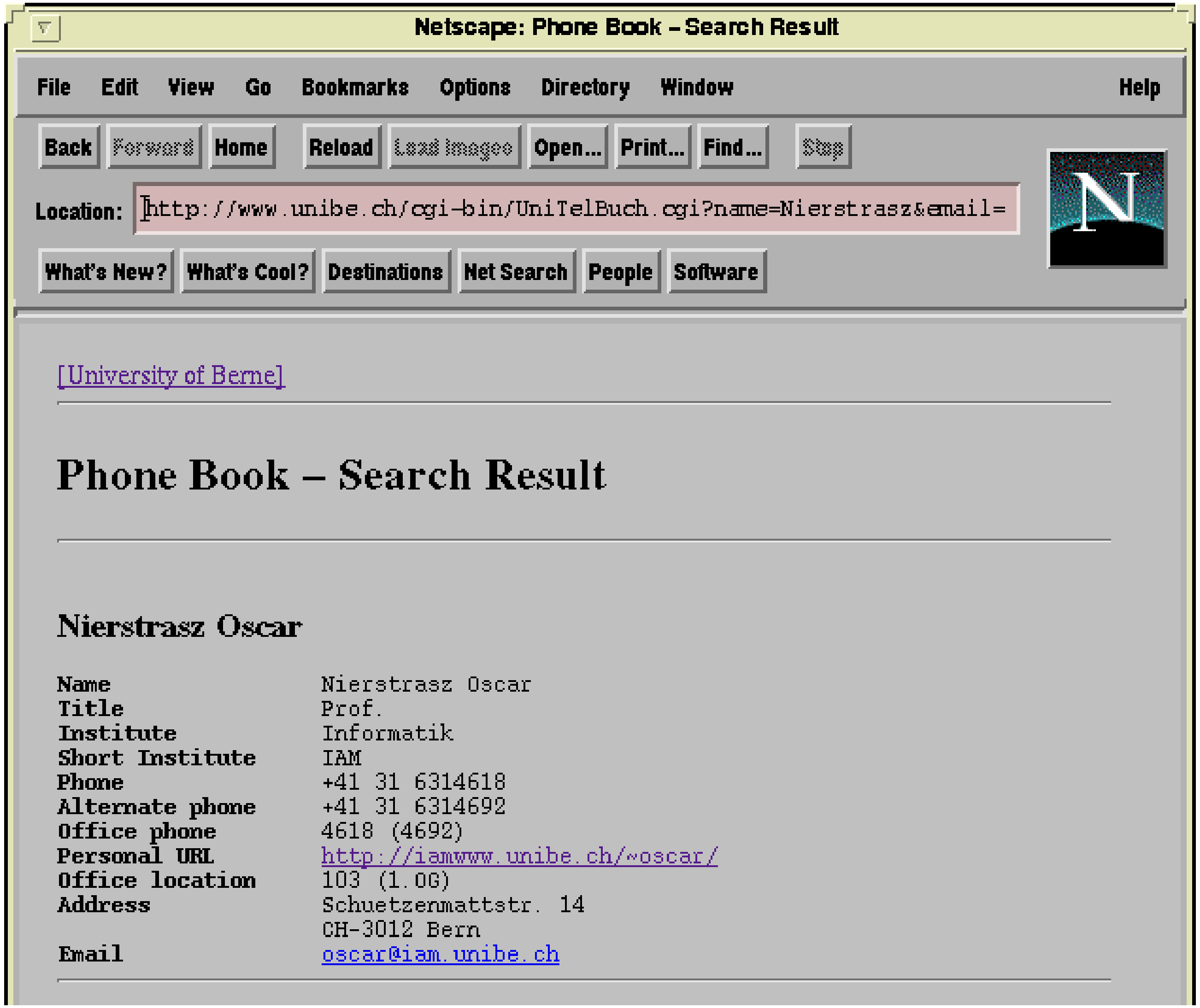
- ❑ Name spaces, modules, objects, multiple inheritance, exceptions
- ❑ “Everything is an object”

A taste of Python

```
oscar@pogo 1: python
Python 1.4 (Jun 4 1997) [GCC 2.7.2]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> 1+2                      # Python can also be used interactively
3
>>> 7/3
2
>>> x, y = 7.0, 3            # NB: tuple assignment
>>> x/y
2.33333333333
>>> "hello world"            # Show the “official” representation
'hello world'
>>> hi = 'hello\nworld'
>>> hi
'hello\012world'
>>> print hi                 # Show the “pretty” string representation of hi
hello
world
>>> hi = hi[:6] + "there"    # Construct new string using slice, and rebind hi
>>> print hi                 # Old value of hi is garbage collected
hello
there
```

```
>>> hi[6] = ' '                # Oops -- strings are immutable!
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: can't assign to this subscripted object
>>> print "%s %s" % (hi[:5], hi[7:])
hello here

>>> range(0,10)                # Generate a list of numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reduce(lambda x, y: x+y, range(0,10))
45
>>> reduce(lambda x, y: x+y, ['hello', 'there'])
'hellothere'

>>> phone = { 'office' : 4618, 'fax' : 3965, 'sec' : 4692 }
>>> phone['fax'] = 3355        # NB: lists and dictionaries are mutable!
>>> phone.keys()
['office', 'fax', 'sec']
>>> phone.values()
[4618, 3355, 4692]
>>> phone.has_key('home')
0
>>> len(phone)
3
```

The Uni Berne on-line Phone Book



Location: http://www.unibe.ch/cgi-bin/UniTelBuch.cgi?name=Nierstrasz&email=

[University of Berne]

Phone Book – Search Result

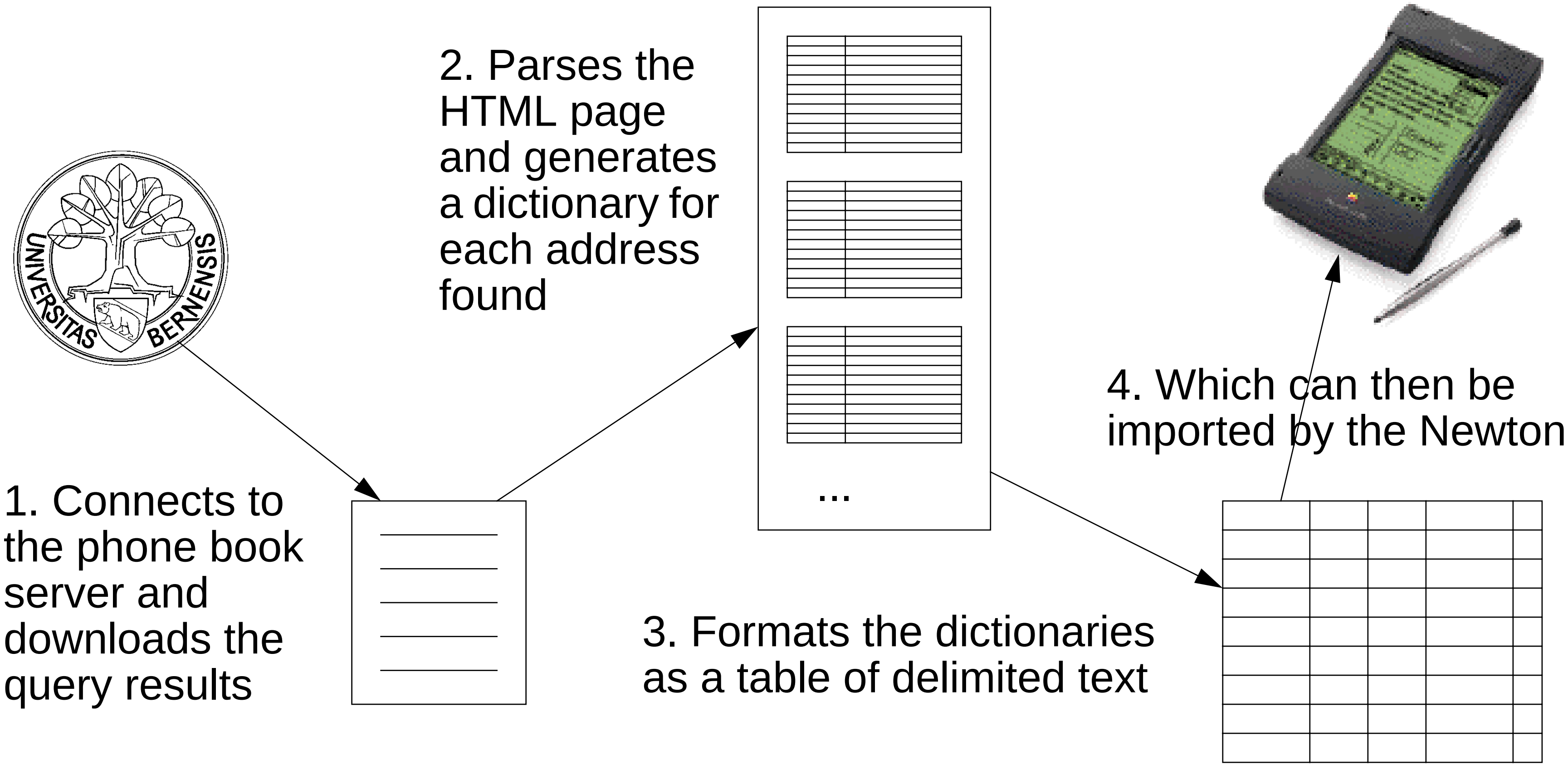
Nierstrasz Oscar

Name	Nierstrasz Oscar
Title	Prof.
Institute	Informatik
Short Institute	IAM
Phone	+41 31 6314618
Alternate phone	+41 31 6314692
Office phone	4618 (4692)
Personal URL	http://iamwww.unibe.ch/~oscar/
Office location	103 (1.OG)
Address	Schuetzenmattstr. 14 CH-3012 Bern
Email	oscar@iam.unibe.ch

Gluing Web Objects

The University's Web Phone Service is nice, but is not ideal for interchanging information with, for example, the Newton MessagePad 2000's Names application.

We would like to script a tool that:



1. Connects to the phone book server and downloads the query results

2. Parses the HTML page and generates a dictionary for each address found

3. Formats the dictionaries as a table of delimited text

4. Which can then be imported by the Newton

The ubtb script interface

```
#! /home/scgstat/Software/python1.4/bin/python
"""
ubtb --- interface to Uni Berne Telephone Book
Usage: ubtb [-dt|-nt] <name> ...
Returns either delimited text (-dt) or normal text (-nt)
(c) Oscar Nierstrasz 1997
"""

import sys                           # System module, for arguments, stderr etc.
from string import split, join       # Some basic string functions

def main():

   parsepage = ParsePage()           # Instantiate a function object

   format = lambda page: showFields(selFields, page)  # default format
   results = []                      # Start with empty list of dictionaries
   for arg in sys.argv[1:]:          # Pick up the script arguments
      if arg == "-dt":               # Toggle the format function to use
         format = lambda page: delText(selFields, page)
      elif arg == "-nt":
         format = lambda page: showFields(selFields, page)
      else:                          # Convert the query results to dictionaries
         results = results + parsepage(getpage(arg))
   format(results)                   # And print them out!
```

Talking to an HTTP server

```
def getpage(name):
   """get an HTML query results for "name" from the
   Uni Berne Phone Book web server"""

   from urllib import urlopen                        # The http equivalent of open()
   ubtb = "http://www.unibe.ch/cgi-bin/UniTelBuch.cgi"
   try:
      name = join(split(name, ' '), '+')             # Replace blanks by ’+’ signs
      url = urlopen("%s?name=%s" % (ubtb,name))      # Supply arguments to CGI script
   except:
      sys.stderr.write("Can't open " + ubtb)
      sys.exit(1)                                    # Exit with error code to shell
   page = url.read()                                 # Read the whole page
   url.close()                                       # Cf. file close
   return page                                       # Return the entire string
```

The HTML results

Now we need to extract the (key,value) pairs from the web page.

```
<DL COMPACT></DL><head>
<title>Phone Book - Search Result</title>
</head>
<body>
<a href="/">[University of Berne]</a>
<hr>
<h1>Phone Book - Search Result</h1>
<pre>
<hr>
</pre><h3>Nierstrasz Oscar</h3><pre>
<strong>Name</strong>                Nierstrasz Oscar
<strong>Title</strong>               Prof.
<strong>Institute</strong>           Informatik
<strong>Short Institute</strong>     IAM
<strong>Phone</strong>               +41 31 6314618
<strong>Alternate phone</strong>     +41 31 6314692
<strong>Office phone</strong>        4618 (4692)
<strong>Personal URL</strong>        <a href="http://iamwww.unibe.ch/~oscar/">http://iamwww. ...
<strong>Office location</strong>     103 (1.OG)
<strong>Address</strong>             Schuetzenmattstr. 14
<strong></strong>                    CH-3012 Bern
<strong>Email</strong>               <a href="mailto:oscar@iam.unibe.ch">oscar@iam.unibe.ch</a>
<hr>
</pre>
<a href="/Adm/Adm.html">Webmaster of the University of Berne</a>
</body>
```

A page parsing function object

The regular expression package provides us with the parsing functionality we need.

Each regular expression must be “compiled” (i.e., into a state machine) before it can be used. Rather than compiling our regular expressions each time we parse a page, we use a function object that compiles them just once, when it is constructed:

```
class ParsePage:
   def __init__(self):
      """initialize a ParsePage function object"""
      import regex
      # Recognize (key, value) pairs:
      self.getFields = regex.compile("^<strong>\([^<]*\)<\/strong>[ \t]*\(.*\)$")
      # Get rid of the HTML anchors surrounding text:
      self.stripAnchor = regex.compile("^<a href=[^>]+>\([^<]+\)<\/a>$")
```

The parts we wish to extract are surrounded by `\( ... \)` *pairs.*

Parsing the HTML

```
class ParsePage:
   ...
   def __call__(self, page):
      """parse output from phone book and return a list of dictionaries"""
      results = []                       # Will hold the list of dictionaries
      gf = self.getFields                # Make some short, local names
      sa = self.stripAnchor
      for line in split(page, "\n"):     # Split the page into lines
         # start a new dictionary at start of new address
         if line[:10] == "</pre><h3>":
            dict = {}                    # Make a new, empty dictionary
            results.append(dict)         # Add it to the end of the list
         else:
            if gf.match(line) > 0:
               (key,val) = gf.group(1,2) # Extract the (key,val) pair
               if sa.match(val) > 0:     # Strip away any HTML anchors
                  val = sa.group(1)      # Extract just the URL
               if key == "":             # An empty key means a continued line
                  key = prevkey          # from the previous key
                  dict[key] = "%s, %s" % (dict[key],val)
               else:
                  dict[key] = val
               prevkey = key             # Remember the key
      return results
```

Formatting

We select the fields that interest us:

```
selFields = [ 'Title', 'Name', 'Institute', 'Address',
      'Phone', 'Alternate phone', 'Email', 'Personal URL', ]
```

The vanilla formatter:

```
def showFields(fields, dictList):         # Print the selected fields
   padding = 20                           # Space reserved for field names
   for dict in dictList:
      for field in fields:
         if dict.has_key(field):          # Print nothing if a field is missing
            print pad(field+":",padding),
            print dict[field]
      print

def pad(s,n):
   """pad a string to a given length"""
   if len(s) < n:
      return s + ' ' * (n - len(s))      # Append a string of blanks
   else:
      return s
```

Vanilla formatting

```
% ubtb "oscar nierstrasz"
Title:                      Prof.
Name:                       Nierstrasz Oscar
Institute:                  Informatik
Address:                    Schuetzenmattstr. 14, CH-3012 Bern
Phone:                      +41 31 6314618
Alternate phone:            +41 31 6314692
Email:                      oscar@iam.unibe.ch
Personal URL:               http://iamwww.unibe.ch/~oscar/
```

Converting dictionaries to lists

We need to convert each dictionary into a list of values for the selected fields.

```
def lookup(dict, keys):
   """lookup up a list of keys in a dictionary,
   returning the list of values"""
   return map(lambda k, d=dict: getField(d,k), keys)


def getField(dict,key):
   """lookup keys in a dictionary, returning an empty string
   if the key is not present (instead of raising an exception)"""
   if dict.has_key(key):
      return dict[key]
   else:
      return ""
```

```
>>> lookup({'a':'A', 'b':'B', 'c':'C'}, ['a', 'b', 'z'])
➪ ['A', 'B', '']
```

Generating delimited text

Now we can apply our formatting function to the list of fields, and to the list of dictionaries

```
def delText(fields, dictList):
   """print selected fields of a list of dictionaries as delimited text.
   Print an empty string if a field is missing."""
   # NB: nested function
   def printList(list):
      """print list of fields, separated by tabs,
      and each surrounded by quotes"""
      print '"%s"' % join(list, '"\t"')
   printList(fields)
   # convert each dictionary to a list of selected fields:
   fieldList = map(lambda dict, fields=fields: lookup(dict, fields), dictList)
   map(printList, fieldList)
```

Note that lambdas do not “capture” names in the local scope, so we must pass them in as default arguments!

Delimited Text

Don’t forget to call main:

```
if __name__ == "__main__":          # If called as a script, call main()
   main()                           # Otherwise, i.e., if imported, do nothing
```

And finally we can generate the delimited text for the Newton:

```
% ubtb -dt "hanspeter bieri" "horst bunke" "gerhard jaeger" "oscar nierstrasz"
"Title"  "Name"                 "Institute"    "Address"      ...
"Prof."  "Bieri Hanspeter"      "Informatik"   "Neubrueckstr. 10, CH-3012 Bern" ...
"Prof."  "Bunke Horst"          "Informatik"   "Neubrueckstr. 10, CH-3012 Bern" ...
"Prof."  "Jaeger Gerhard"       "Informatik"   "Neubrueckstr. 10, CH-3012 Bern" ...
"Prof."  "Nierstrasz Oscar"     "Informatik"   "Schuetzenmattstr. 14, CH-3012 ...
```

Summary

You should know the answers to these questions:

- ☐ How does "scripting" differ from "programming"?
- ☐ What happens when you "import" a module in Python?
- ☐ What is the difference between "`import`" and "`from` ... `import`"?
- ☐ What happens when you evaluate "`x = x + y`"?
- ☐ Does it matter if x is a number or a string? A user-defined object?
- ☐ How are run-time type errors handled?
- ☐ When can objects be garbage-collected?

Can you answer the following questions?

- ✎ *Why are strings immutable in Python if other kinds of lists are mutable?*
- ✎ *How would you construct a dictionary from a list of (key, value) pairs?*
- ✎ *What would this program look like without using* `lambda` *and* `map`*?*
- ✎ *How many ways can you think of to replace blanks in a string by "*`+`*" signs?*
- ✎ *How would you write a script that produces an HTML index of all pages at a web site that are reachable from its home page?*

13. Summary, Trends, Research ...

- ❑ Summary: functional, logic, object-oriented and scripting languages
- ❑ Open Systems Development
- ❑ Research directions ...
  - ☞ http://iamwww.unibe.ch/~scg/

Functional Languages

Good for:

- ❑ equational reasoning
- ❑ declarative programming

Bad for:

- ❑ OOP
- ❑ explicit concurrency
- ❑ run-time efficiency (although constantly improving)

Trends:

- ❑ standardization: Haskell, “ML 2000”
- ❑ extensions (concurrency, objects): Facile, “ML 2000”, UFO ...

Lambda Calculus

Good for:

- simple, operational foundation for sequential programming languages

Bad for:

- programming

Trends:

- concurrent, distributed calculi (e.g., π calculus, “join” calculus ...)

Type Systems

Good for:

- catching type errors
- documenting interfaces
- formalizing and reasoning about domains of functions and objects

Bad for:

- self-modifying programs

Trends:

- automatic type inference
- reasoning about concurrency and other side effects

Polymorphism

Good for:

- ❑ parametric good for generic containers
- ❑ subtyping good for frameworks (generic clients)
- ❑ overloading syntactic convenience (classes in gopher, overloading in Java)
- ❑ coercion convenient, but may obscure meaning

Bad for:

- ❑ local reasoning
- ❑ optimization

Trends:

- ❑ combining subtyping, polymorphism and overloading
- ❑ exploring alternatives to subtyping (“matching”)

Denotational Semantics

Good for:

- ❑ formally and unambiguously specifying languages
- ❑ sequential languages

Bad for:

- ❑ modelling concurrency and distribution

Trends:

- ❑ “Natural Semantics” (inference rules vs. equations)
- ❑ concurrent, distributed calculi

Logic Programming

Good for:

- searching (expert systems, graph & tree searching ...)
- symbolic interpretation

Bad for:

- debugging
- modularity

Trends:

- constraints
- concurrency

Object-Oriented Languages

Good for:

- data abstraction
- modelling real-world “objects”
- developing reusable frameworks
- dynamic binding; various forms of polymorphism

Bad for:

- learning (steep learning curve)
- understanding (hard to keep systems well-structured)
- semantics (no agreement)

Trends:

- extensions to existing paradigms (functional, logic, constraint ...)
- extensions to concurrency, distribution
- object-oriented “scripting” (Perl, Python, JavaScript, ActiveX)

Scripting Languages

Good for:

- rapid prototyping
- high-level programming
- reflection; on-the-fly generation and evaluation of programs
- gluing components from different environments

Bad for:

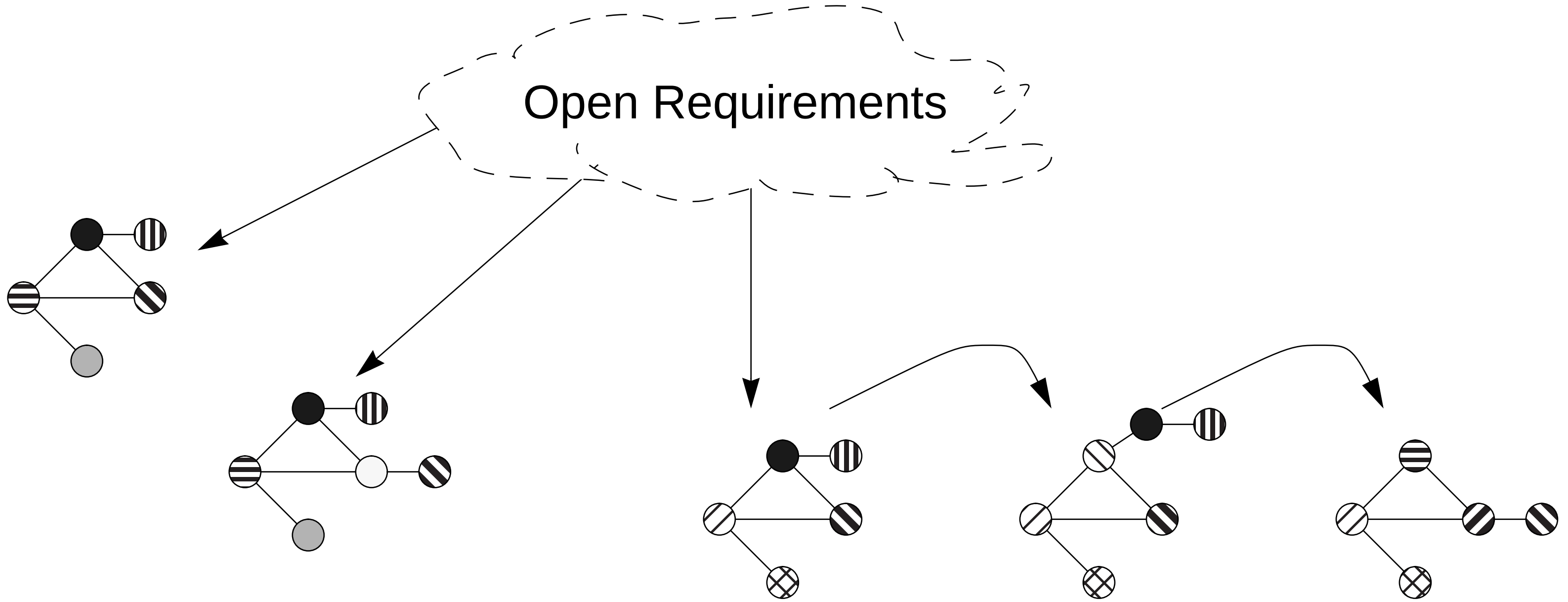
- type-checking; reasoning about program correctness
- performance-critical applications

Trends:

- replacing programming as main development paradigm
- scriptable applications
- graphical “builders” instead of languages

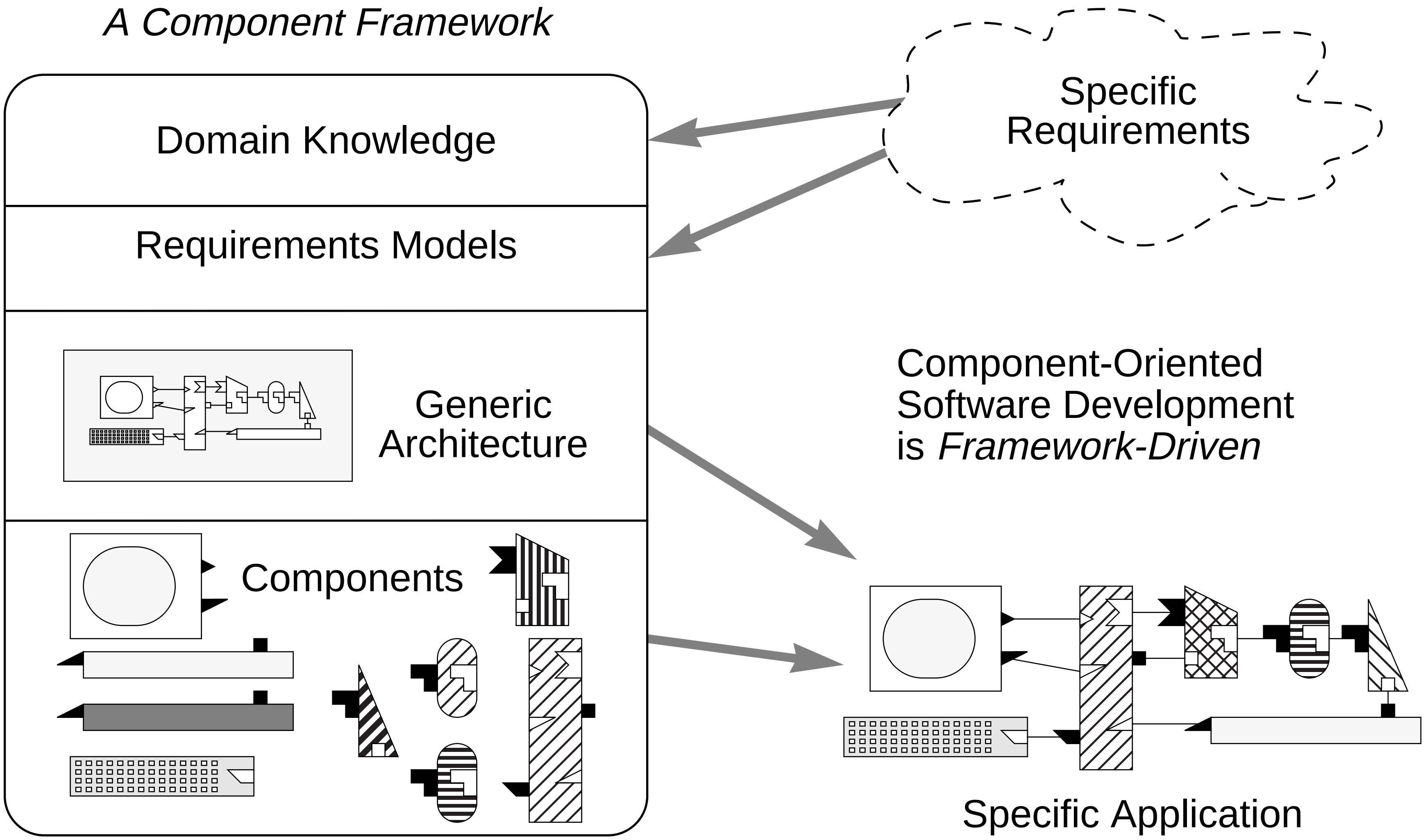
Open Systems are Families of Applications

Open systems undergo changing requirements:



An individual system may either be an *instance* of a generic *family* of applications, or a *snapshot in time* of a changing application.

Component-Oriented Development



A Component Framework

Domain Knowledge

Requirements Models

Generic Architecture

Components

Specific Requirements

Component-Oriented Software Development is *Framework-Driven*

Specific Application

Research Issues

1. Languages:
    - ☞ How to specify components, architectures and frameworks?
    - ☞ How to specify applications as compositions?

2. Tools:
    - ☞ How to represent and manage framework knowledge?
    - ☞ How to visually present and manipulate software components?

3. Frameworks:
    - ☞ What are good examples of components and generic architectures?

4. Methods:
    - ☞ How to drive application development from frameworks?
    - ☞ How to iteratively develop and evolve component frameworks?