Programmiersprachen

Prof. O. Nierstrasz

Sommersemester 2003

Table of Contents

1. Programming Languages

Lecturer:	Prof. Oscar Nierstrasz Schützenmattstr. 14/103
Tel:	031 631 4618
Email:	Oscar.Nierstrasz@iam.unibe.ch
Assistants:	Gabriela Arévalo, Marc-Philippe Horvath
WWW:	www.iam.unibe.ch/~scg/Teaching/

Sources

Text:

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

Other Sources:

- Bjarne Stroustrup, *The C++ Programming Language* (Special Edition), Addison Wesley, 2000.
- *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, 1989, pp 359-411.
- Clocksin and Mellish, *Programming in Prolog*, Springer Verlag, 1981.

Schedule

1.	03 - 25	Introduction
2.	04 - 01	Systems programming
3.	04 - 08	Multi-paradigm programming
4.	04 - 15	Stack-based programming
5.	04 - 22	Functional programming
6.	04 - 29	Type systems
7.	05 - 06	Lambda calculus
8.	05 - 13	Fixed points
9.	05 - 20	Programming language semantics
10.	05 - 27	Logic programming
11.	06 - 03	Applications of logic programming
12.	06 - 10	Piccola — A Small Composition Language
13.	06 - 17	Summary, Trends, Research
	06 - 24	*Final exam*

What is a Programming Language?

- A formal language for describing computation?
- A "user interface" to a computer?
- Syntax + semantics?
- Compiler, or interpreter, or translator?
- A tool to support a programming paradigm?

"A programming language is a notational system for describing computation in a machine-readable and human-readable form."

— *Louden*

What is a Programming Language? (II)

The thesis of this course:

> *A programming language is a tool for developing executable models for a class of problem domains.*

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?

...

Themes Addressed in this Course ...

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?

Generations of Programming Languages

1GL: machine codes

2GL: symbolic assemblers

3GL: (machine independent) imperative languages (FORTRAN, Pascal, C ...)

4GL: domain specific application generators

Each generation is at a higher level of abstraction

How do Programming Languages Differ?

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

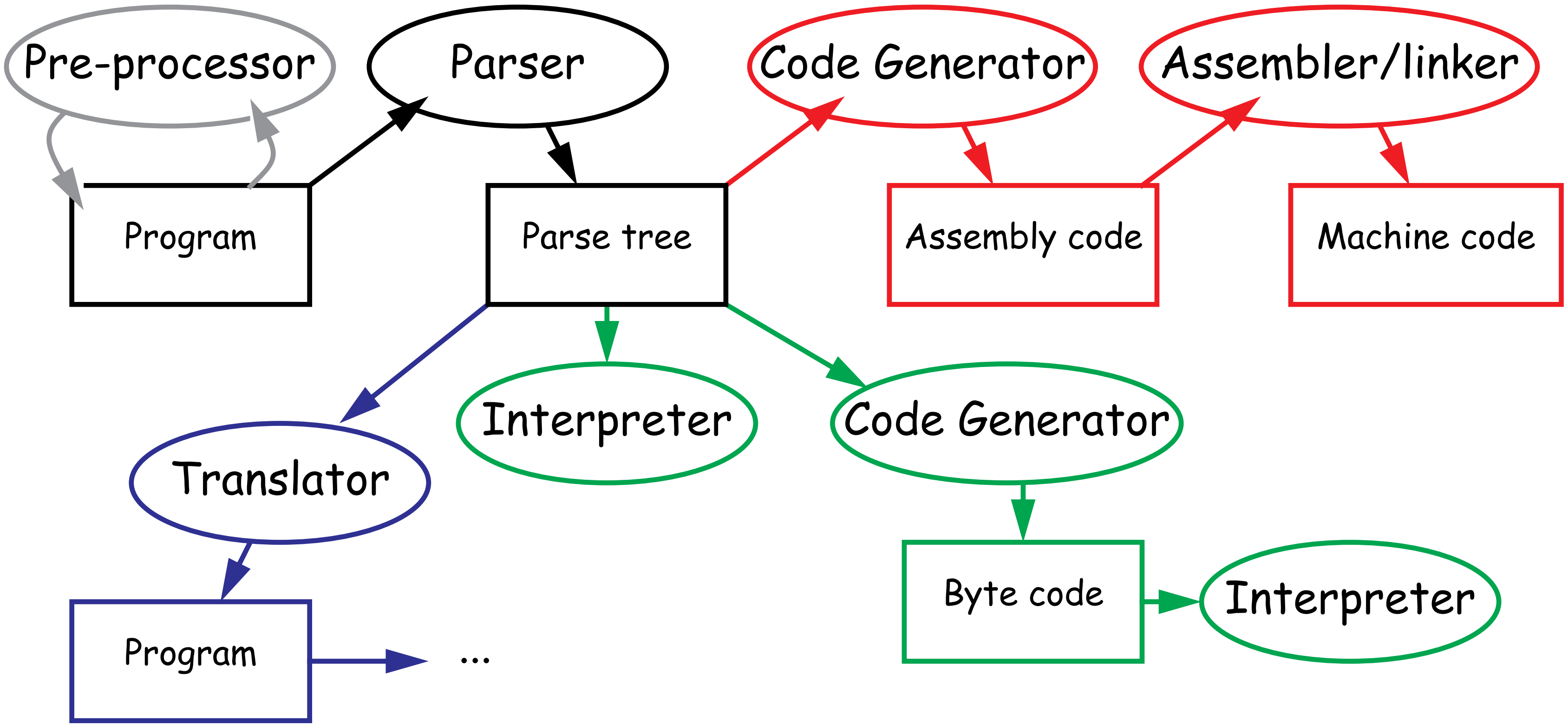
A programming language is a problem-solving tool.

Imperative style:	program = algorithms + data *good for decomposition*
Functional style:	program = functions $\circ$ functions *good for reasoning*
Logic programming style:	program = facts + rules *good for searching*
Object-oriented style:	program = objects + messages *good for encapsulation*

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 (3GL is invented)
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 (a proposal)
1970	Prolog	*logic* programming is born
1972	C	*the* systems programming language
1975	Pascal, Scheme	two teaching languages

1978	CSP	Concurrency matures
1978	FP	Backus' proposal
1983	Smalltalk-80, Ada	OOP is reinvented
1984	Standard ML	FP becomes mainstream (?)
1986	C++, Eiffel	OOP is reinvented (again)
1988	CLOS, Oberon, Mathematica	
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

...

Fortran ...

Innovations

- *Symbolic notation* for subroutines and functions
- Assignments to variables of complex expressions
- DO loops
- Comments
- Input/output formats
- Machine-independence

Successes

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

"Hello World" in FORTRAN

```
      PROGRAM HELLO
      DO 10, I=1,10
      PRINT *,'Hello World'
10    CONTINUE
      STOP
      END
```

All examples from the ACM "Hello World" project:

www2.latech.edu/~acm/HelloWorld.shtml

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

...

ALGOL 60 ...

Innovations

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

Successes

- Highly influenced design of other PLs but never displaced FORTRAN

"Hello World" in BEALGOL

```
BEGIN
FILE F (KIND=REMOTE);
EBCDIC ARRAY E [0:11];
REPLACE E BY "HELLO WORLD!";
WHILE TRUE DO
  BEGIN
  WRITE (F, *, E);
  END;
END.
```

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 (!)

“Hello World” in COBOL

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000300 DATE-WRITTEN.   02/05/96        21:04.
000400* AUTHOR BRIAN COLLINS
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
001000 DATA DIVISION.
001100 FILE SECTION.
100000 PROCEDURE DIVISION.
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400    DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500    DISPLAY "HELLO, WORLD." LINE 15 POSITION 10.
100600    STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800    EXIT.
```

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*

"Hello World" in RPG

```
H
FSCREEN O F 80 80          CRT
C                    EXCPT
OSCREEN E 1
O                               12 'HELLO WORLD!'
```

"Hello World" in SQL

```
CREATE TABLE HELLO (HELLO CHAR(12))
UPDATE HELLO
  SET HELLO = 'HELLO WORLD!'
SELECT * FROM HELLO
```

PL/1

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

"Hello World" in PL/1

```
HELLO:    PROCEDURE OPTIONS (MAIN);

            /* A PROGRAM TO OUTPUT HELLO WORLD */
            FLAG = 0;

LOOP:     DO WHILE (FLAG = 0);
            PUT SKIP DATA('HELLO WORLD!');
          END LOOP;

END HELLO;
```

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)

```
10 print "Hello World!"
20 goto 10
```

...

Interactive Languages ...

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)

```
'HELLO WORLD'
```

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

```
  OUTPUT = 'Hello World!'
END
```

...

Special-Purpose Languages ...

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*

```
(DEFUN HELLO-WORLD ()
    (PRINT (LIST 'HELLO 'WORLD)))
```

Functional Languages

ISWIM (If you See What I Mean)

- Peter Landin (1966) — 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*"

"Hello World" in Functional Languages

SML

```
print("hello world!\n");
```

Haskell

```
hello() = print "Hello World"
```

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

"Hello World" in Prolog

```
% HELLO WORLD. Works with Sbp (prolog)

hello :-
printstring("HELLO WORLD!!!!").

printstring([]).
printstring([H|T]) :- put(H), printstring(T).
```

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

```
Begin
   while 1 = 1 do begin
      outtext ("Hello World!");
      outimage;
   end;
End;
```

- **Smalltalk** was developed by Xerox PARC (early 1970s) to drive graphic workstations

```
Transcript show:'Hello World';cr
```

...

Object-Oriented Languages ...

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

```
  echo "Hello, World!"
```

- **HyperTalk** (1987) was developed at Apple to script HyperCard stacks

```
on OpenStack
  show message box
  put "Hello World!" into message box
end OpenStack
```

...

Scripting Languages ...

- **TCL** (1990) developed as embedding language and scripting language for X windows applications (via Tk)

```
puts "Hello World "
```

- **Perl** (~1990) became de facto web scripting language

```
print "Hello, World!\n";
```

...

Scripting Languages ...

Innovations

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

Successes

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

What you should know!

- ✎ 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 C a "Third Generation Language"?
- ✎ What is a "Fourth Generation Language"?

Can you answer these questions?

- ✎ *Why are there so many programming languages?*
- ✎ *Why are FORTRAN and COBOL still important programming languages?*
- ✎ *Which language should 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. Systems Programming

Overview

- C Features
- Memory layout
- Declarations and definitions
- Working with Pointers

Reference:

- Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice Hall, 1978.
- Kernighan and Plauger, *The Elements of Programming Style, McGraw-Hill*, 1978.

What is C?

C was designed as a *general-purpose language* with a very *direct mapping* from data types and operators to machine instructions.

- *cpp* (C pre-processor) used for expanding macros and inclusion of declaration "header files"
- explicit *memory allocation* (no garbage collection)
- memory manipulation through *pointers*, pointer arithmetic and typecasting
- used as *portable*, high-level assembler

C Features

Developed in 1972 by Dennis Ritchie and Brian Kernighan as a *systems language* for Unix on the PDP-11. A successor to B [Thompson, 1970], in turn derived from BCPL.

C preprocessor:	file inclusion, conditional compilation, macros
Data types:	char, short, int, long, double, float
Type constructors:	pointer, array, struct, union
Basic operators:	arithmetic, pointer manipulation, bit manipulation ...
Control abstractions:	if/else, while/for loops, switch, goto ...
Functions:	call-by-value, side-effects through pointers
Type operations:	typedef, sizeof, explicit type-casting and coercion

"Hello World" in C

Pre-processor directive: include declarations for standard i/o library

A comment

Function definition: there is always a "main" function

```
#include <stdio.h>
/* My first C program! */
int main(void)
{
  printf("hello world!\n");
  return 0;
}
```

A string constant: an array of 14 (not 13!) chars

Symbols

C programs are built up from *symbols*:

Names:	{ alphabetic or underscore } followed by { alphanumerics or underscores } `main, IOStack, _store, x10`
Keywords:	`const, int, if, ...`
Constants:	`"hello world", 'a', 10, 077, 0x1F, 1.23e10`
Operators:	`+, >>, *, &`
Punctuation:	`{, }, ,`

Keywords

C has a large number of reserved words:

Control flow:	`break, case, continue, default, do, else, for, goto, if, return, switch, while`
Declarations:	`auto, char, const, double, extern, float, int, long, register, short, signed, static, struct, typedef, union, unsigned, void`
Expressions:	`sizeof`

Operators (same as Java)

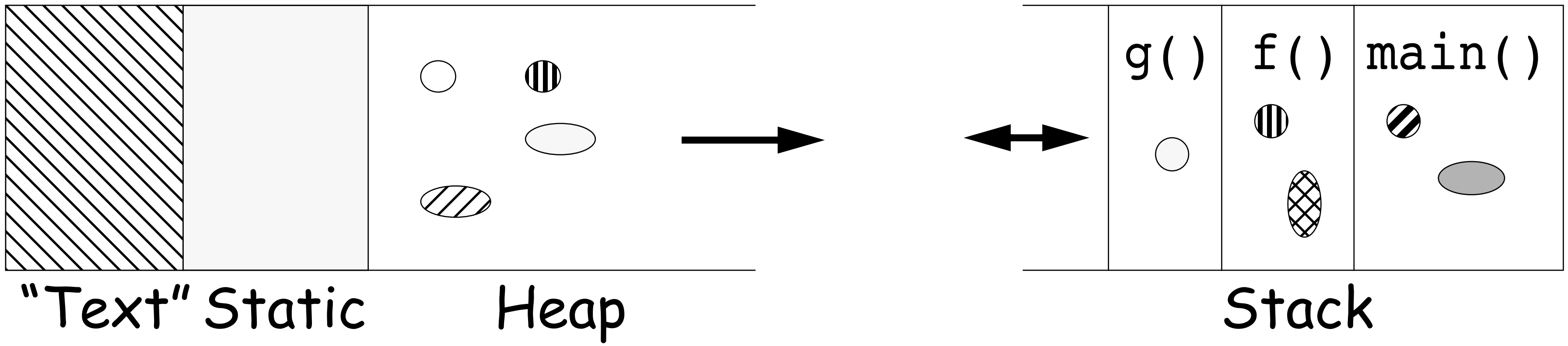
```
int a, b, c;
double d;
float f;
a = b = c = 7;          assignment:           a == 7; b == 7; c == 7
a = (b == 7);           equality test:        a == 1 (7 == 7)
b = !a;                 negation:             b == 0 (!1)
a = (b>=0)&&(c<10);     logical AND:          a == 1 ((0>=0)&&(7<10))
a *= (b += c++);        increment:            a == 7; b == 7; c == 8
a = 11 / 4;             integer division:     a == 2
b = 11 % 4;             remainder:            b == 3
d = 11 / 4;                                   d == 2.0 (not 2.75!)
f = 11.0 / 4.0;                               f == 2.75
a = b|c;                bitwise OR:           a == 11 (03|010)
b = a^c;                bitwise XOR:          b == 3 (013^010)
c = a&b;                bitwise AND:          c == 3 (013&03)
b = a<<c;               left shift:           b == 88 (11<<3)
a = (b++,c--);          comma operator:       a == 3; b == 89; c == 2
b = (a>c)?a:c;          conditional operator:     b == 3 ((3>2)?3:2)
```

C Storage Classes

You must explicitly manage storage space for data

Static	❑ static objects exist for the *entire life-time* of the process
Automatic	❑ only live *during function invocation* on the "run-time stack"
Dynamic	❑ dynamic objects live between calls to `malloc` and `free` ❑ their lifetimes typically *extend beyond their scope*

Memory Layout



The address space consists of (at least):

Text:	executable program text (not writable)
Static:	static data
Heap:	dynamically allocated global memory (grows upward)
Stack:	local memory for function calls (grows downward)

Where is memory?

```
#include <stdio.h>

static int stat=0;
void dummy() { }

int main(void)
{
   int local=1;
   int *dynamic = (int*) malloc(sizeof(int),1);

   printf("Text is here: %u\n", (unsigned) dummy); /* function pointer */
   printf("Static is here: %u\n", (unsigned) &stat);
   printf("Heap is here: %u\n", (unsigned) dynamic);
   printf("Stack is here: %u\n", (unsigned) &local);
}
```

```
Text is here: 7604
Static is here: 8216
Heap is here: 279216
Stack is here: 3221223448
```

Declarations and Definitions

Variables and functions must be either declared or defined *before* they are used:

- A *declaration* of a variable (or function) announces that the variable (function) exists and is defined somewhere else.

```
extern char *greeting;
void hello(void);
```

- A *definition* of a variable (or function) causes storage to be allocated

```
char *greeting =
      "hello world!\n";
void hello(void)
{
  printf(greeting);
}
```

Header files

C does not provide modules — instead one should break a program into *header* files containing declarations, and *source* files containing definitions that may be separately compiled.

hello.h

```
extern char *greeting;
void hello(void);
```

hello.c

```
#include <stdio.h>

char *greeting = "hello world!\n";

void hello(void)
{
   printf(greeting);
}
```

Including header files

Our main program may now *include* declarations of the separately compiled definitions:

helloMain.c

```
#include "hello.h"

int main(void)
{
   hello();
   return 0;
}
```

cc -c helloMain.c	*compile to object code*
cc -c hello.c	*compile to object code*
cc helloMain.o hello.o -o helloMain	*link to executable*

Makefiles

You could also compile everything together:

cc helloMain.c hello.c -o helloMain

Or you could use a *makefile* to manage *dependencies*:

```
helloMain : helloMain.c hello.h hello.o
	cc helloMain.c hello.o -o $@

...
```

✎ *"Read the manual"*

C Arrays

Arrays are *fixed sequences* of *homogeneous elements*.

- `Type a[n];` defines a one-dimensional array `a` in a contiguous block of `(n*sizeof(Type))` bytes
- `n` must be a compile-time *constant*
- Arrays bounds run from *0 to n-1*
- *Size cannot vary* at run-time
- They can be initialized at compile time:

```
int eightPrimes[8] =
        { 2, 3, 5, 7, 11, 13, 17, 19 };
```

- But *no range-checking* is performed at run-time:

```
eightPrimes[8] = 0; /* disaster! */
```

Pointers

A *pointer* holds the *address* of another variable:

```
int i = 10;
int *ip = &i; /* assign the address of i to ip */
```

Use them to access and update variables:	`*ip = *ip + 1;`
Array variables behave like pointers to their first element	`int *ep = eightPrimes;`
Pointers can be treated like arrays:	`ep[7] = 23;`
But have different sizes:	`sizeof(eightPrimes) == 32)` `sizeof(ep) == 4)`
You may increment and decrement pointers:	`ep = ep+1;`
*Declare a pointer to an unknown data type as void**	`void *vp = ep;`
But typecast it properly before using it!	`((int*)vp)[6] = 29;`

Strings

A *string* is a pointer to a NULL-terminated (i.e., '\0') character array:

`char *cp;`	*uninitialized string (pointer to a char)*
`char *hi = "hello";`	*initialized string pointer*
`char hello[6] = "hello";`	*initialized char array*
`cp = hello;`	*cp now points to hello[]*
`cp[1] = 'u';`	*cp and hello now point to "hullo"*
`cp[4] = NULL;`	*cp and hello now point to "hull"*

✎ *What is sizeof(hi)? sizeof(hello)?*

Pointer manipulation

Copy string s1 to buffer s2:

```
void strCopy(char s1[], char s2[])
{
   int i = 0;
   while (s1[i] != '\0') {     /* Assume s1 is NULL-terminated! */
      s2[i] = s1[i];           /* assume s2 is big enough! */
      i++;
   }
   s2[i] = '\0';
}
```

More idiomatically (!):

```
void strCopy2(char *s1, char *s2)
{
   while (*s2++ = *s1++);      /* fails only when NULL is reached */
}
```

Function Pointers

```
int ascii(char c) { return((int) c); } /* cast */

void applyEach(char *s, int (*fptr)(char)) {
  char *cp;
  for (cp = s; *cp; cp++)
    printf("%c -> %d\n", *cp, fptr(*cp));
}

int main(int argc, char *argv[]) {
  int i;
  for (i=1;i<argc;i++)
    applyEach(argv[i], ascii);
  return 0;
}
```

```
./fptrs abcde
a -> 97
b -> 98
c -> 99
d -> 100
e -> 101
```

Working with pointers

Problem: read an arbitrary file, and print out the lines in reverse order.

Approach:

- Check the file size
- Allocate enough memory
- Read in the file
- Starting from the end of the buffer
  - Convert each newline ('\n') to a NULL ('\0')
  - printing out lines as you go
- Free the memory.

Argument processing

```
int main(int argc, char* argv[])
{
  int i;
  if (argc<1) {
    fprintf(stderr, "Usage: lrev <file> ...\n");
    exit(-1);
  }
  for (i=1;i<argc;i++) {
    lrev(argv[i]);
  }
  return 0;
}
```

Using pointers for side effects

Return pointer to file contents or NULL (error code)
Set bytes to file size

```
char* loadFile(char *path, int *bytes)
{
  FILE *input;
  struct stat fileStat; /* see below ... */
  char *buf;
  *bytes = 0; /* default return val */
  if (stat(path, &fileStat) < 0) { /* POSIX std */
    return NULL; /* error-checking vs exceptions */
  }
  *bytes = (int) fileStat.st_size;
...
```

Memory allocation

NB: Error-checking code left out here for readability ...

```
    ...
    buf = (char*) malloc(sizeof(char)*((*bytes)+1));
    ...
    input = fopen(path, "r");
    ...
    int n = fread(buf, sizeof(char), *bytes, input);
    ...
    buf[*bytes] = '\0'; /* terminate buffer */
    fclose(input);
    return buf;
}
```

Pointer manipulation

```
void lrev(char *path)
{
  char *buf, *end;
  int bytes;
  buf = loadFile(path, &bytes);
...
  end = buf + bytes - 1; /* last byte of buffer */
  if ((*end == '\n') && (end >= buf)) {
    *end = '\0';
  }
...
```

✎ *What if bytes = 0?*

Pointer manipulation ...

```
/* walk backwards, converting lines to strings */

  while (end >= buf) {
    while ((*end != '\n') && (end >= buf))
      end--;
    if ((*end == '\n') && (end >= buf))
      *end = '\0';
    puts(end+1);
  }
  free(buf);
}
```

✎ *Is this algorithm correct? How would you prove it?*

Built-In Data Types

The precision of built-in data types may depend on the machine architecture!

Data type	*No. of bits*	*Minimal value*	*Maximal value*
`signed char`	*8*	*-128*	*127*
`signed short`	*16*	*-32768*	*32767*
`signed int`	*16 / 32*	*-32768 / -2147483648*	*32767 / 214748647*
`signed long`	*32*	*-2147483648*	*214748647*
`unsigned char`	*8*	*0*	*255*
`unsigned short`	*16*	*0*	*65535*
`unsigned int`	*16 / 32*	*0*	*65535 / 4294967295*
`unsigned long`	*32*	*0*	*4294967295*

Built-In Data Types ...

Data type	No. of bytes	Min. exponent	Max. exponent	Decimal accuracy
`float`	4	-38	+38	6
`double`	8	-308	+308	15
`long double`	8 / 10	-308 / -4932	+308 / 4932	15 / 19

User Data Types

Data structures are defined as C "structs".

In /usr/include/sys/stat.h:

```
struct stat {
   dev_t      st_dev;      /* inode's device */
   ino_t      st_ino;      /* inode's number */
   mode_t     st_mode;     /* inode protection mode */
   nlink_t    st_nlink;    /* number of hard links */
   uid_t      st_uid;      /* user ID of the file's owner */
   gid_t      st_gid;      /* group ID of the file's group */
   ...
   off_t      st_size;     /* file size, in bytes */
   int64_t    st_blocks;   /* blocks allocated for file */
   ...
};
```

Typedefs

Type names can be assigned with the typdef command:

```
typedef   long long   int64_t;
typedef   int64_t     quad_t;
typedef   quad_t      off_t;     /* file offset */
```

Observations

- C can be used as either a high-level or low-level language
  - generally used as a *"portable assembler"*
- C gives you complete freedom
  - requires great *discipline* to use correctly
- *Pointers* are the greatest source of errors
  - off-by-one errors
  - invalid assumptions
  - failure to check return values

Obfuscated C

A fine tradition since 1984 ...

```
#define iv 4
#define v ;(void
#define XI(xi)int xi[iv*'V'];
#define L(c,l,i)c(){d(l);m(i);}
#include <stdio.h>
int*cc,c,i,ix='\t',exit(),X='\n'*'\d';XI(VI)XI(xi)extern(*vi[])(),(*
signal())();char*V,cm,D['x'],M='\n',I,*gets();L(MV,V,(c+='d',ix))m(x){v)
signal(X/'I',vi[x]);}d(x)char*x;{v)write(i,x,i);}L(MC,V,M+I)xv(){c>=i?m(
c/M/M+M):(d(&M),m(cm));}L(mi,V+cm,M)L(md,V,M)MM(){c=c*M%X;V-=cm;m(ix);}
LXX(){gets(D)||(vi[iv])();c=atoi(D);while(c>=X){c-=X;d("m");}V="ivxlcdm"
+iv;m(ix);}LV(){c-=c;while((i=cc[*D=getchar()])>-I)i?(c?(c<i&&l(-c-c,
"%d"),l(i,"+%d")):l(i,"(%d")):(c&&l(M,")"),l(*D,"%c")),c=i;c&&l(X,")"),l
(-i,"%c");m(iv-!(i&I));}L(ml,V,'\f')li(){m(cm+!isatty(i=I));}ii(){m(c=cm
= ++I)v)pipe(VI);cc=xi+cm++;for(V="jWYmDEnX";*V;V++)xi[*V^' ']=c,xi[*V++]
=c,c*=M,xi[*V^' ']=xi[*V]=c>>I;cc[-I]-=ix v)close(*VI);cc[M]-=M;}main(){
(*vi)();for(;v)write(VI[I],V,M));}l(xl,lx)char*lx;{v)printf(lx,xl)v)
fflush(stdout);}L(xx,V+I,(c-=X/cm,ix))int(*vi[])()={ii,li,LXX,LV,exit,l,
d,l,d,xv,MM,md,MC,ml,MV,xx,xx,xx,xx,MV,mi};
```

A C Puzzle

✎ *What does this program do?*

```
char f[] = "char f[] = %c%s%c;%cmain() {printf(f, 34,
f, 34, 10, 10);}%c";
main() {printf(f, 34, f, 34, 10, 10);}
```

What you should know!

- ✎ What is a header file for?
- ✎ What are declarations and definitions?
- ✎ What is the difference between a char* and a char[]?
- ✎ How do you allocate objects on the heap?
- ✎ Why should every C project have a makefile?
- ✎ What is sizeof(“abcd”)?
- ✎ How do you handle errors in C?
- ✎ How can you write functions with side-effects?
- ✎ What happens when you increment a pointer?

Can you answer these questions?

- Where can you find the system header files?
- What's the difference between c++ and ++c?
- How do malloc and free manage memory?
- How does malloc get more memory?
- What happens if you run: free("hello")?
- How do you write portable makefiles?
- What is sizeof(&main)?
- What trouble can you get into with typecasts?
- What trouble can you get into with pointers?

3. Multiparadigm Programming

Overview

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- Templates and STL

References:

- Bjarne Stroustrup, *The C++ Programming Language* (Special Edition), Addison Wesley, 2000.

Essential C++ Texts

- Stanley B. Lippman and Josee LaJoie, *C++ Primer*, Third Edition, Addison-Wesley, 1998.
- Scott Meyers, *Effective C++*, 2d ed., Addison-Wesley, 1998.
- James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
- David R. Musser, Gilmer J. Derge and Atul Saini, *STL Tutorial and Reference Guide*, 2d ed., Addison-Wesley, 2000.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

What is C++?

A *"better C"* that supports:

- Object-oriented programming (classes & inheritance)
- Generic programming (templates)
- Programming-in-the-large (namespaces, exceptions)
- Systems programming (thin abstractions)
- Reuse (large standard class library)

C++ vs C

Most C programs are also C++ programs.

Nevertheless, good C++ programs usually do not resemble C:

- avoid macros (use `inline`)
- avoid pointers (use references)
- avoid `malloc` and `free` (use `new` and `delete`)
- avoid arrays and `char*` (use `vector`s and `string`s) ...
- avoid `struct`s (use `class`es)

C++ encourages a different style of programming:

- avoid procedural programming
  - ☞ *model your domain* with classes and templates

"Hello World" in C++

Include standard iostream classes

A C++ comment

```
#include <iostream>
// My first C++ program!
int main(void)
{
  cout << "hello world!" << endl;
  return 0;
}
```

cout is an instance of ostream

operator overloading (two different argument types!)

C++ Design Goals

"C with Classes" designed by Bjarne Stroustrup in early 1980s:

- Originally a translator to C
  - Initially difficult to debug and inefficient
- Mostly *upward compatible* extension of C
  - "As close to C as possible, but no closer"
  - Stronger type-checking
  - Support for object-oriented programming
- Run-time efficiency
  - Language primitives close to machine instructions
  - *Minimal cost* for new features

C++ Features

C with Classes	Classes as structs Inheritance; virtual functions Inline functions
C++ 1.0 (1985)	Strong typing; function prototypes `new` and `delete` operators
C++ 2.0	Local classes; protected members Multiple inheritance
C++ 3.0	Templates Exception handling
ANSI C++ (1998)	Namespaces RTTI

Java and C++ — Similarities and Extensions

Similarities:

- primitive data types (in Java, platform independent)
- syntax: control structures, exceptions ...
- classes, visibility declarations (`public`, `private`)
- multiple constructors, `this`, `new`
- types, type casting (safe in Java, not in C++)

Java Extensions:

- garbage collection
- standard abstract machine
- standard classes (came later to C++)
- packages (now C++ has namespaces)
- final classes

Java Simplifications

- no pointers — just *references*
- no functions — can declare `static` methods
- no global variables — use `public static` variables
- no *destructors* — garbage collection and `finalize`
- no linking — dynamic class loading
- no header files — can define `interface`
- no *operator overloading* — only method overloading
- no member initialization lists — call `super` constructor
- no preprocessor — `static final` constants and automatic inlining
- no multiple inheritance — implement multiple interfaces
- no structs, unions, enums — typically not needed
- no *templates* — but generics will likely be added ...

New Keywords

In addition the keywords inherited from C, C++ adds:

Exceptions	`catch, throw, try`
Declarations:	`bool, class, enum, explicit, export, friend, inline, mutable, namespace, operator, private, protected, public, template, typename, using, virtual, volatile, wchar_t`
Expressions:	`and, and_eq, bitand, bitor, compl, const_cast, delete, dynamic_cast, false, new, not, not_eq, or, or_eq, reinterpret_cast, static_cast, this, true, typeid, xor, xor_eq`

Comments

Two styles:

```
/*
 * C-style comment pairs are generally used
 * for longer comments that span several lines.
 */

// C++ comments are useful for short comments
```

Use // comments exclusively within functions so that any part can be commented out using comment pairs.

References

A *reference* is an *alias* for another variable:

```
int i = 10;
int &ir = i;
ir = ir + 1;    // increment i
```

Once initialized, references *cannot be changed*.

References are especially useful in *procedure calls* to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing

```
void refInc(int &n)
{
  n = n+1; // increment the variable n refers to
}
```

References vs Pointers

References should be *preferred to pointers* except when:

- manipulating dynamically allocated objects
  - ☞ `new` returns an object pointer
- a variable must range over a *set* of objects
  - ☞ use a pointer to walk through the set

C++ Classes

C++ classes may be instantiated either *automatically* (on the stack):

```
MyClass oVal;           // constructor called
                        // destroyed when scope ends
```

or *dynamically* (in the heap)

```
MyClass *oPtr;          // uninitialized pointer

oPtr = new MyClass;     // constructor called
                        // must be explicitly deleted
```

Constructors and destructors

Constructors can make use of *member initialization lists:*

```
class MyClass {
private:
  string _name;
public:
  MyClass(string name) : _name(name) { // constructor
    cout << "create " << name << endl;
  }
  ~MyClass() {                        // destructor
    cout << "destroy " << _name << endl;
  }
};
```

C++ classes can specify cleanup actions in *destructors*

Automatic and dynamic destruction

```
MyClass& start() {   // returns a reference
  MyClass a("a");                    // automatic
  MyClass *b = new MyClass("b");     // dynamic
  return *b;         // returns a reference (!) to b
}                    // a goes out of scope

void finish(MyClass& b) {
  delete &b;         // need pointer to b
}

finish(start());
```

```
create a
create b
destroy a
destroy b
```

Orthodox Canonical Form

Most of your classes should look like this:

```
class myClass {
public:
  myClass(void);                    // default constructor
  myClass(const myClass& copy); // copy constructor
    ...                             // other constructors
  ~myClass(void);                   // destructor
  myClass& operator=(const myClass&); // assignment
    ...       // other public member functions
private:
    ...
};
```

Why OCF?

If you don't define these four member functions, C++ will generate them:

- default constructor
  - will call default constructor for each data member
- destructor
  - will call destructor of each data member
- copy constructor
  - will *shallow copy* each data member
  - pointers will be copied, not the objects pointed to!
- assignment
  - will *shallow copy* each data member

Example: A String Class

We would like a String class that protects C-style strings:

- strings are indistinguishable from char pointers
- string updates may cause memory to be corrupted

Strings should support:

- creation and destruction
- initialization from char arrays
- copying
- safe indexing
- safe concatenation and updating
- output
- length, and other common operations ...

A Simple String.h

```
class String
{
   friend ostream& operator<<(ostream&, const String&);
public:
   String(void);                          // default constructor
   ~String(void);                         // destructor
   String(const String& copy);            // copy constructor
   String(const char*s);                  // char* constructor
   String& operator=(const String&);      // assignment

   inline int length(void) const { return ::strlen(_s); }
   char& operator[](const int n) throw(exception);
   String& operator+=(const String&) throw(exception); // concatenation
private:
   char *_s; // invariant: _s points to a null-terminated heap string
   void become(const char*) throw(exception); // internal copy function
};
```

Default Constructors

Every constructor should *establish the class invariant*:

```
String::String(void)
{
    _s = new char[1];     // allocate a char array
    _s[0] = '\0';         // NULL terminate it!
}
```

The *default constructor* for a class is called when a new instance is declared without any initialization parameters:

```
String anEmptyString; // call String::String()
String stringVector[10]; // call it ten times!
```

Destructors

The String destructor must *explicitly* free any memory allocated by that object.

```
String::~String (void)
{
  delete [] _s; // delete the char array
}
```

Every new must be matched somewhere by a delete!

- use `new` and `delete` for objects
- use `new[]` and `delete[]` for arrays!

Copy Constructors

Our String copy constructor must create a *deep copy*:

```
String::String(const String& copy)
{
  become(copy._s); // call helper
}

void String::become(const char* s) throw (exception)
{
  _s = new char[::strlen(s) + 1];
  if (_s == 0) throw(logic_error("new failed"));
  ::strcpy(_s, s);
}
```

A few remarks ...

- If we do not define our own copy constructor, copies of Strings will *share the same representation!*
  - Modifying one will modify the other!
  - Destroying one will invalidate the other!
- If we do not declare `copy` as `const`, we will not be able to construct a copy of a *const* `String`!
- If we declare `copy` as `String` rather than `String&`, *a new copy will be made* before it is passed to the constructor!
  - Functions *arguments* are always *passed by value* in C++
  - The "value" of a pointer is a pointer!
- The abstraction boundary is a class, *not an object*. Within a class, all private members are visible (as is copy._s)

Other Constructors

Class constructors may have arbitrary arguments, as long as their signatures are unique and unambiguous:

```
String::String(const char* s)
{
  become(s);
}
```

Since the argument is not modified, we can declare it as `const`. This will allow us to construct String instances from constant char arrays.

Assignment Operators

Assignment is different from the copy constructor because *an instance already exists*:

```
String& String::operator=(const String& copy)
{
  if (this != &copy) {          // take care!
    delete [] _s;
    become(copy._s);
  }
  return *this; // NB: a reference, not a copy
}
```

A few more remarks ...

- Return `String&` rather than `void` so the result can be *used in an expression*
- Return `String&` rather than `String` so *the result won't be copied!*
- `this` is a pseudo-variable whose value is a pointer to the current object
  - ☞ so `*this` is the *value* of the current object, which is *returned by reference*

Implicit Conversion

When an argument of the “wrong” type is passed to a function, the C++ compiler looks for a constructor that will convert it to the “right” type:

```
str = "hello world";
```

is implicitly converted to:

```
str = String("hello world");
```

Operator Overloading

Not only assignment, but other useful operators can be "overloaded" provided their signatures are unique:

```
char&
String::operator[] (const int n) throw(exception)
{
  if ((n<0) || (length()<=n)) {
    throw(logic_error("array index out of bounds"));
  }
  return _s[n];
}
```

NB: a non-const reference is returned, so can be used as an lvalue in an assignment.

Overloadable Operators

The following operators may be overloaded:

<table>
<tr><th colspan="8">Overloadable Operators</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>-</td><td>!</td><td>,</td><td>=</td><td><</td><td>></td><td><=</td><td>>=</td></tr>
<tr><td>++</td><td>--</td><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td>*=</td></tr>
<tr><td><<=</td><td>>>=</td><td>[]</td><td>()</td><td>-></td><td>->*</td><td>new</td><td>delete</td></tr>
</table>

NB: arity and precendence are fixed by C++

Friends

We would like to be able to write:

```
cout << String("TESTING ... ") << endl;
```

But:

- ☞ It can't be a member function of ostream, since we can't extend the standard library.
- ☞ It can't be a member function of `String` since the target is `cout`.
- ☞ But it must have access to `String`'s private data

So ... we need a binary *function* << that takes a `cout` and a `String` as arguments, and is a *friend* of `String`.

Friends ...

We declare:

```
class String
{
  friend ostream&
        operator<<(ostream&, const String&);
  ...
};
```

And define:

```
ostream&
operator<<(ostream& outStream, const String& s)
{
  return outStream << s._s;
}
```

What are Templates?

A template is a *generic specification* of a function or a class, *parameterized* by one or more types used within the function or class:

- functions that only assume basic operations of their arguments (comparison, assignment ...)
- "container classes" that do little else but hold instances of other classes

Templates are essentially glorified macros

- like macros, they are compiled only when instantiated (and so are *defined exclusively in header files*)
- unlike macros, templates are not expanded literally, but may be intelligently processed by the C++ compiler

Function Templates

The following declares a generic `min()` function that will work for arbitrary, comparable elements:

```
template <class Item>
inline const Item&
min (const Item& a, const Item& b)
{
  return (a<b) ? a : b;
}
```

Templates are automatically instantiated by need:

```
cout << "min(3,5) = " << min(3,5) << endl;
// instantiates: inline const int& min(int&, int&);
```

Class Templates

Class templates are declared just like function templates:

```
template <class First, class Second>
class pair {
public:
  First first;
  Second second;
  pair(const First& f, const Second& s) :
    first(f), second(s) {}
};
```

Using Class Templates

Template classes are instantiated by binding the formal parameter:

```
typedef pair<int, char*> MyPair;

MyPair myPair = MyPair(6, "I am not a number");

cout << myPair.first << " sez "
    << myPair.second << endl;
```

Typedefs are a convenient way to bind names to template instances.

Standard Template Library

STL is a general-purpose C++ library of generic algorithms and data structures.

1. *Containers* store collections of objects
   - ☞ vector, list, deque, set, multiset, map, multimap
2. *Iterators* traverse containers
   - ☞ random access, bidirectional, forward/backward ...
3. *Function Objects* encapsulate functions as objects
   - ☞ arithmetic, comparison, logical, and user-defined ...
4. *Algorithms* implement generic procedures
   - ☞ search, count, copy, random_shuffle, sort, ...
5. *Adaptors* provide an alternative interface to a component
   - ☞ stack, queue, reverse_iterator, ...

An STL Line Reverser

```
#include <iostream>
#include <stack>                        // STL stacks
#include <string>                       // Standard strings

void rev(void)
{
   typedef stack<string> IOStack;       // instantiate the template
   IOStack ioStack;                     // instantiate the template class
   string buf;

   while (getline(cin, buf)) {
      ioStack.push(buf);
   }
   while (ioStack.size() != 0) {
      cout << ioStack.top() << endl;
      ioStack.pop();
   }
}
```

What we didn't have time for ...

- virtual member functions, pure virtuals
- public, private and multiple inheritance
- default arguments, default initializers
- method overloading
- `const` declarations
- enumerations
- smart pointers
- static and dynamic casts
- template specialization
- namespaces
- RTTI

...

What you should know!

- ✎ What new features does C++ add to C?
- ✎ What does Java remove from C++?
- ✎ How should you use C and C++ commenting styles?
- ✎ How does a reference differ from a pointer?
- ✎ When should you use pointers in C++?
- ✎ Where do C++ objects live in memory?
- ✎ What is a member initialization list?
- ✎ Why does C++ need destructors?
- ✎ What is OCF and why is it important?
- ✎ What's the difference between delete and delete[]?
- ✎ What is operator overloading?
- ✎ Why are templates like macros?

Can you answer these questions?

- ✎ Why doesn't C++ support garbage collection?
- ✎ Why doesn't Java support multiple inheritance?
- ✎ What trouble can you get into with references?
- ✎ Why doesn't C++ just make deep copies by default?
- ✎ How can you declare a class without a default constructor?
- ✎ Why can objects of the same class access each others private members?
- ✎ Why are templates only defined in header files?
- ✎ How are templates compiled?
- ✎ What is the type of a template?

4. 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

What is 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

Postscript variants

Level 1:

- ❑ the original 1985 PostScript

Level 2:

- ❑ additional support for dictionaries, memory management ...

Display PostScript:

- ❑ special support for screen display

Level 3:

- ❑ the current incarnation with "workflow" support

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 consisting of "regular characters" but which 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 items and divide by 2`

Semantics

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

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 ...

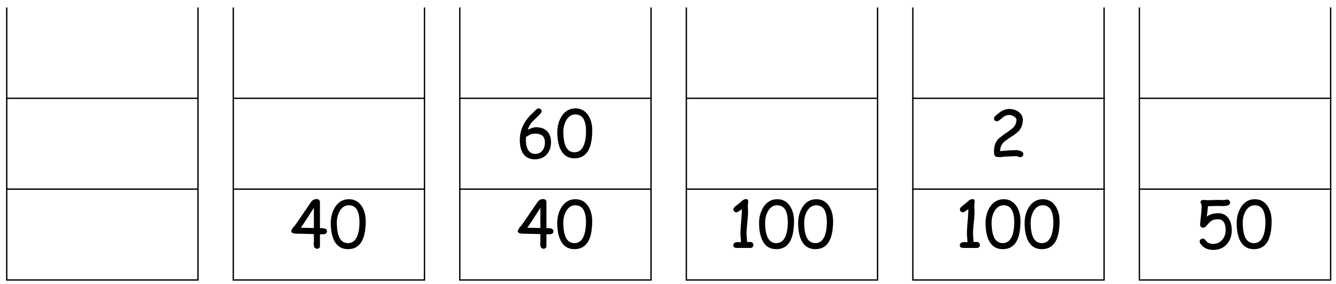
Composite Object Types are copied by *reference*

- 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

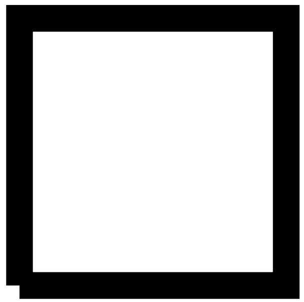
Stack	Op	New Stack	Function
num_1 num_2	`add`	sum	$num_1 + num_2$
num_1 num_2	`sub`	difference	$num_1 - num_2$
num_1 num_2	`mul`	product	$num_1 * num_2$
num_1 num_2	`div`	quotient	num_1 / num_2
int_1 int_2	`idiv`	quotient	integer divide
int_1 int_2	`mod`	remainder	int_1 mod int_2
num den	`atan`	angle	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

and many others ...

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 (100,100)
100 200 lineto         % draw a line to (100,200)
200 200 lineto
200 100 lineto
100 100 lineto
10 setlinewidth        % set width for drawing
stroke                 % draw along current path
showpage               % and display current page
```



Path construction operators

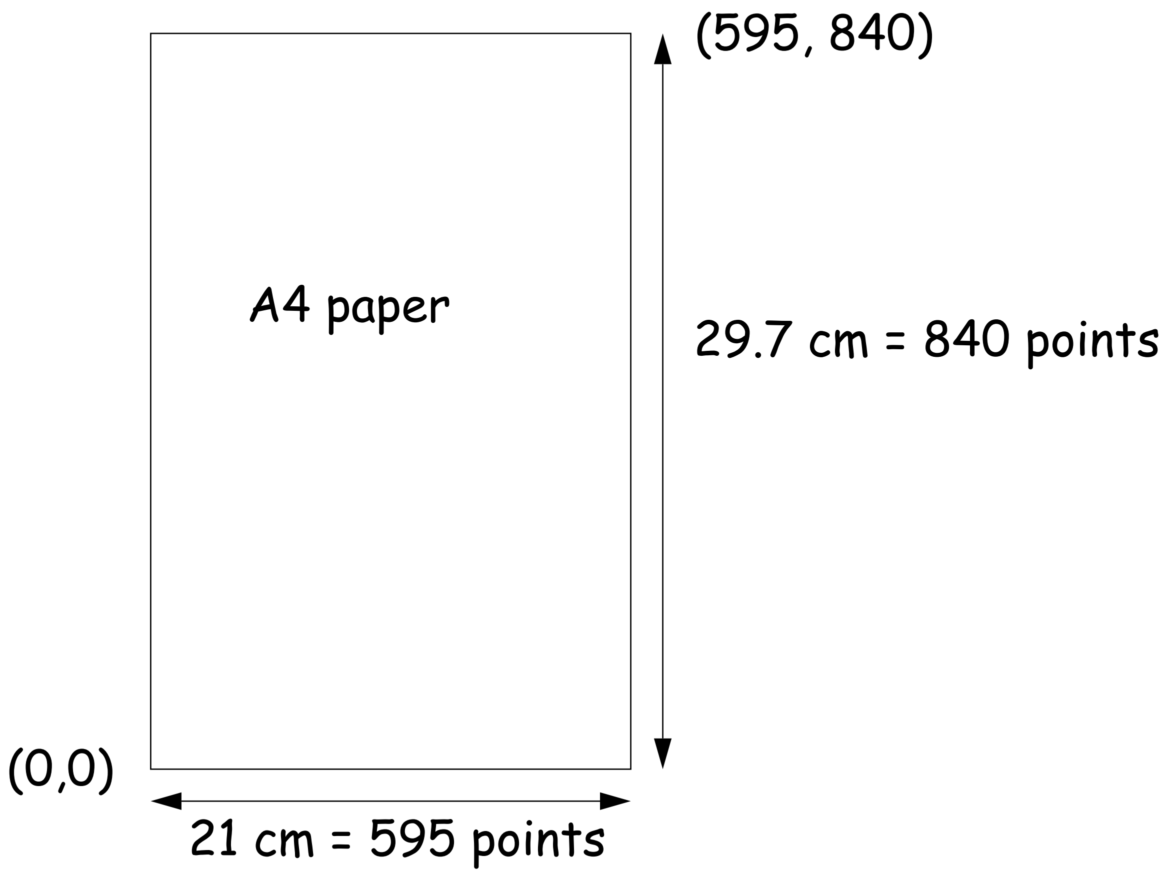
-	`newpath`	-	initialize current path to be empty
-	`currentpoint`	x y	return current coordinates
x y	`moveto`	-	set current point to *(x, y)*
dx dy	`rmoveto`	-	relative moveto
x y	`lineto`	-	append straight line to *(x, y)*
dx dy	`rlineto`	-	relative lineto
x y r ang_1 ang_2	`arc`	-	append counterclockwise 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

Others: arcn, arcto, curveto, rcurveto, flattenpath, ...

Coordinates

Coordinates are measured in *points:*

72 points = 1 inch = 2.54 cm.



"Hello World" in Postscript

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 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

Others: definefont, makefont, FontDirectory, StandardEncoding ....

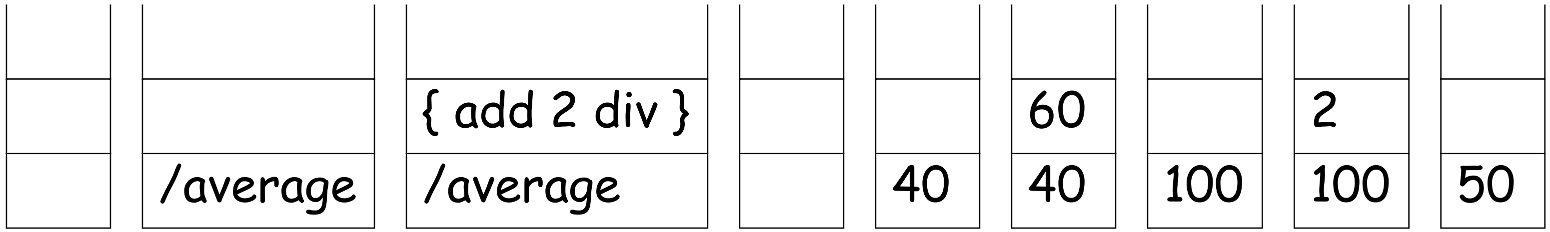
Procedures and Variables

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:

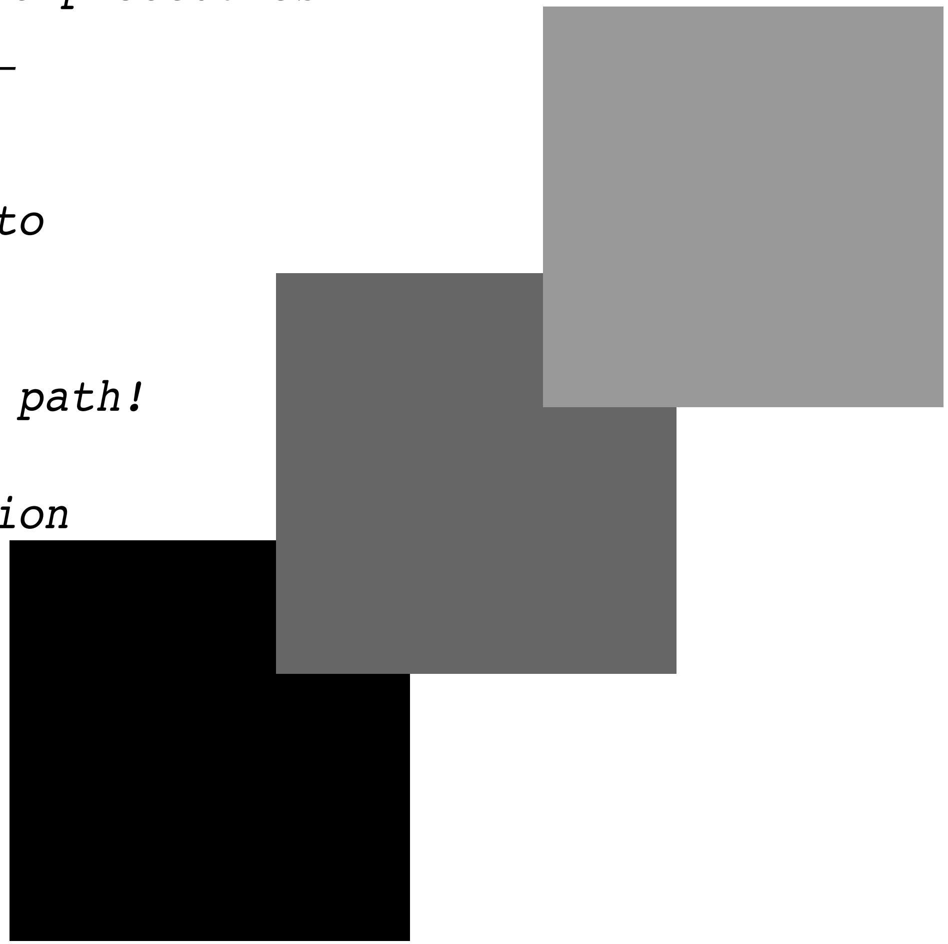
```
/average { add 2 div } def
% bind the name "average" to "{ add 2 div }"
40 60 average
```



A Box procedure

Most PostScript programs consist of 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
```



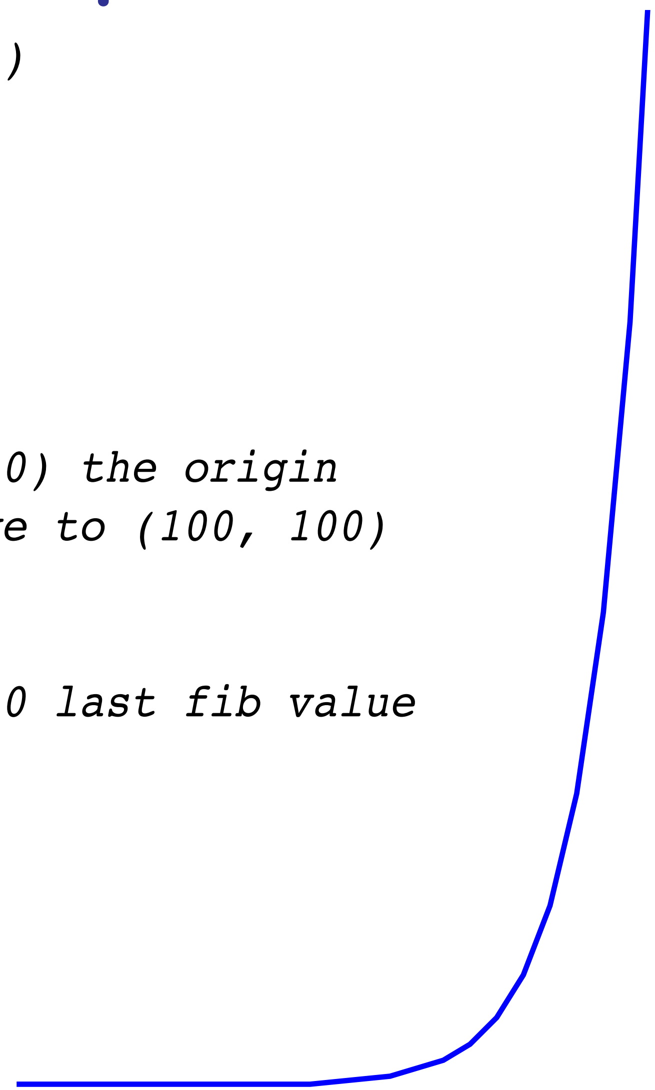
Graphics state and coordinate operators

num	`setlinewidth`	-	set line width
num	`setgray`	-	set colour to gray value (0 = black; 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

gsave saves the current path, gray value, line width and user coordinate system

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
```



Numbers and Strings

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

Factorial

```
/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
```

Factorial ...

```
/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
```

A simple formatter ...

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

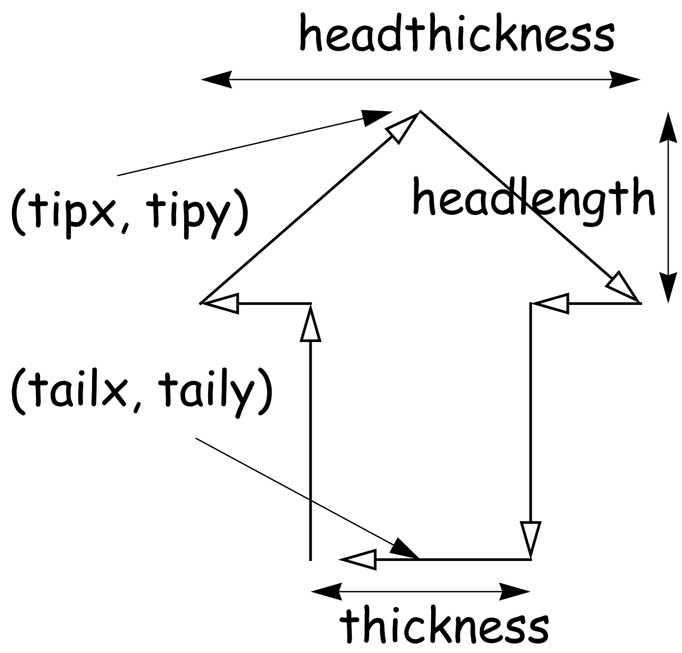
Now is the time for
all good men to
come to the aid of
the party.

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

Using Dictionaries — Arrowheads

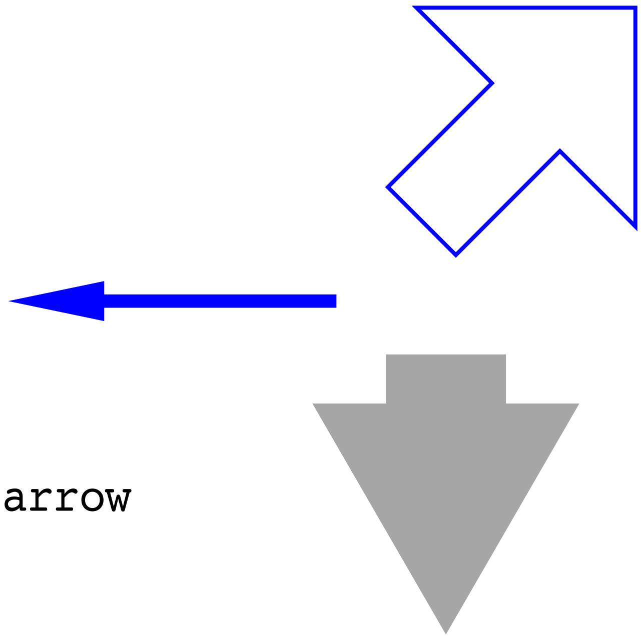
```
/arrowdict 14 dict def                 % make a new dictionary
arrowdict begin
  /mtrx matrix def                     % allocate space for a matrix
end
/arrow {
  arrowdict begin % open the dictionary
    /headlength exch def % grab args
    /halfheadthickness exch 2 div def
    /halfthickness exch 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
```

What you should know!

- ✎ *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 these 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?*

5. 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, 1989, 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.
- Simon Peyton Jones and John Hughes [editors], *Report on the Programming Language Haskell 98 A Non-strict, Purely Functional Language*, February 1999
  - ☞ www.haskell.org

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 ...

A Bit of History

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

Imperative Programming:

☞ Program = Algorithms + Data

Functional Programming:

☞ Program = Functions $\circ$ Functions

What is a Program?

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

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*

What is 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 98 report

"Hello World" in Hugs

```
hello() = print "Hello World"
```

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

Of course, real functional languages are not implemented by syntactic substitution ...

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 be *optimized away* by a modern compiler since it needs only a single run-time stack frame:

```
fact 5  →  fact 5 | fact 4  →  fact 5 | fact 4 | fact 3

     sfac 5  →  sfac 4  →  sfac 3
```

...

Tail Recursion ...

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 = 1 = sfac 1 0`

n > 0: Suppose

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

...

Equational Reasoning ...

Lemma:

For all $n \geq 0$, `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

Haskell support multiple styles for specifying case-based function definitions:

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]`

Using Lists

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 treat other functions as *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]`

NB: `map fac` is a new function that can be applied to lists:

```
mfac = map fac
```

`mfac [1..3]`

➪ `[1, 2, 6]`

Anonymous functions

Anonymous functions can be written as “lambda abstractions”. The function `(\x -> x * x)` behaves exactly like `sqr`:

```
sqr x = x * x

sqr 10                ⇨ 100
(\x -> x * x) 10      ⇨ 100
```

Anonymous functions are first-class values:

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

Curried functions

A <u>Curried function</u> [named after the logician H.B. Curry] *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

plus'(x,y) = x + y        -- normal addition
plus'(1,2) ⇨ 3
```

Understanding Curried functions

```
plus x y = x + y
```

is the same as:

```
plus x = \y -> x+y
```

In other words, plus is *a function of one argument* that *returns a function* as its result.

```
plus 5 6
```

is the same as:

```
(plus 5) 6
```

In other words, we invoke (plus 5), obtaining a function,

```
\y -> 5 + y
```

which we then pass the argument 6, yielding 11.

Using Curried functions

Curried functions are useful because we can bind their argument *incrementally*

```
inc     = plus 1              -- bind first argument to 1
inc 2   ⇨ 3

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

Currying

The following (pre-defined) function takes a binary function as an argument and turns it into a curried function:

```
curry f a b  = f (a, b)

plus(x,y)   = x + y                -- not curried!
inc         = (curry plus) 1

sfac(s, n) = if    n == 0          -- not curried
             then  s
             else  sfac (s*n, n-1)

fac = (curry sfac) 1               -- bind first argument
```

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      = (1,0)
fibPair (n+2)  = (a+b,a)
     where (a,b) = fibPair (n+1)
```

✎ *How would you write a tail-recursive Fibonacci 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)

from 10 ➪ [10,11,12,13,14,15,16,17,....

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..]

Programming lazy lists

Many sequences are naturally implemented as lazy lists.
Note the top-down, declarative style:

```
fibs = 1 : 1 : fibsFollowing 1 1
    where fibsFollowing a b =
      (a+b) : fibsFollowing 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?*

Declarative Programming Style

```
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        = notDivisible primes n
notDivisible (k:ps) n
   | (k*k) > n       = True
   | (mod n k) == 0 = False
   | otherwise       = notDivisible ps n
```

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

What you should know!

- 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 these 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?

6. Type Systems

Overview

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

What is a Type?

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 Types

Values have <u>*static types*</u> defined by the programming language.

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

declared, static type is Applet

static type of *value* is GameApplet

```
Applet myApplet = new GameApplet();
```

actual dynamic type is GameApplet

Static and Dynamic Typing

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.

- *Primitive types:* booleans, integers, floats, chars ...
- *Composite types:* functions, lists, tuples ...

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

- *User-defined types:* enumerations, recursive types, generic types, objects ...

Type Completeness

The Type Completeness Principle:

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

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:

```
Int -> Int -> Int         ≡   Int -> (Int -> Int)
```

and

```
plus 5 6                  ≡   ((plus 5) 6).
```

so:

```
plus::Int->Int->Int       ⇒   plus 5::Int->Int
```

List 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

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 `()`.

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 either:

1. *Named* constructors:

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

2. *Binary* constructors (i.e., starting with ":"):

   ```
   type1 CONOP type2
   ```

Enumeration types

User data types that do not hold any data can model enumerations:

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

Functions over user data types must *deconstruct* the arguments, with one case for each constructor:

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

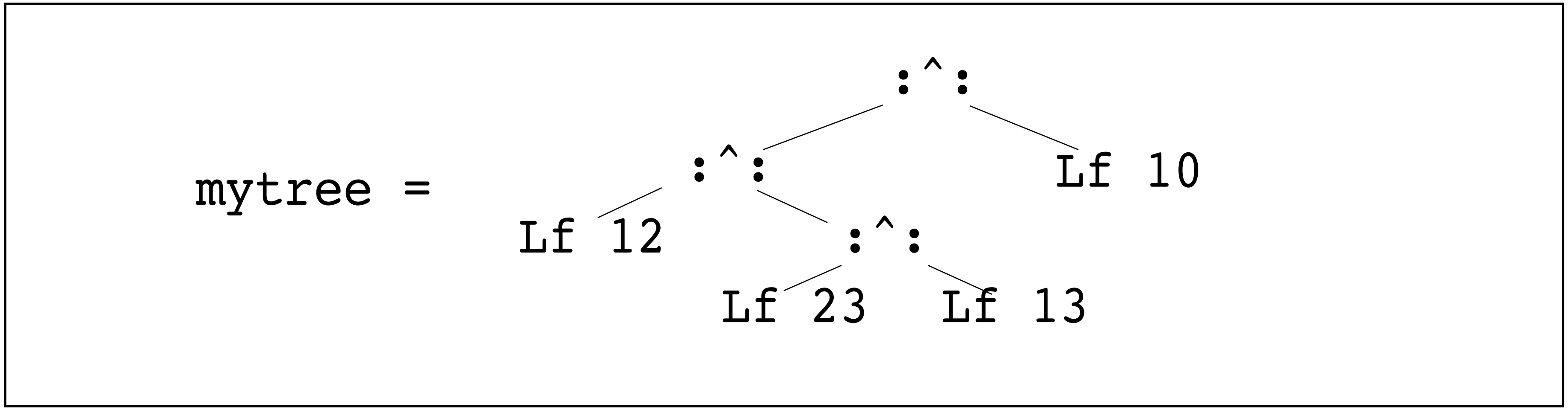
Union types

```
data Temp = Centigrade Float | Fahrenheit Float

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

Recursive Data Types

A recursive data type provides constructors over the type itself:

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

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




```
? :t mytree ⇨ mytree :: Tree Int
```

Using recursive data types

```
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
```

- ✎ *What do these functions do?*
- ✎ *Which function should be more efficient? Why?*
- ✎ *What is (l:) and what does it do?*

Monomorphism

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

- *good* for *type-checking*
- *bad* for writing *generic* code
  - ☞ it is impossible in Pascal to write a generic sort procedure

Polymorphism

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)
```

Type Inference

We can *infer* the type of many expressions by simply examining their structure. Consider:

```
length [ ]      = 0
length (x:xs)   = 1 + length xs
```

Clearly:

```
length :: a -> b
```

Furthermore, *b* is obvious *int*, and *a* is a list, so:

```
length :: [c] -> int
```

We cannot further refine the type, so we are done.

Composing polymorphic types

We can *deduce* the types of expressions using polymorphic functions by simply *binding type variables to concrete types.*

Consider:

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

Then:

```
map length                        :: [[a]] -> [Int]
[ "Hello", "World" ]              :: [[Char]]
map length [ "Hello", "World" ]   :: [Int]
```

Polymorphic Type Inference

Hindley-Milner Type Inference provides an effective algorithm for automatically determining the types of polymorphic functions.

```
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 ]
```

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/void pointer type in Pascal/C*
  - *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 vs overloading

Coercion or overloading — how does one distinguish?

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

✎ *Are there several overloaded + functions, or just one, with values automatically coerced?*

Overloading

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

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

A type class must be *instantiated* to be used:

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

Instantiating overloaded operators

For each overloaded instance a separate definition must be given ...

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

Equality for Data Types

Why not automatically provide equality for all types of values?

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
```

✎ *How would you define equality for the Tree data type?*

NB: all ('elem' ys) xs tests that every x in xs is an element of ys

Equality for Functions

Functions:

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

Determining equality of functions is *undecidable* in general!

What you should know!

- ✎ 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 these 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?
- ✎ How would you define equality for the Tree data type?

7. Introduction to the Lambda Calculus

Overview

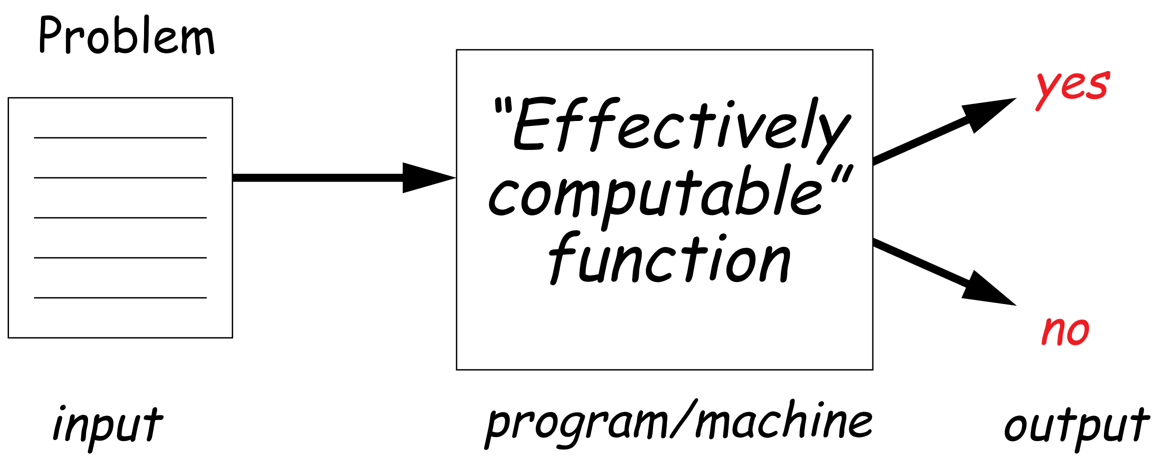
- What is Computability? — Church's Thesis
- Lambda Calculus — operational semantics
- The Church-Rosser Property
- Modelling basic programming constructs

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? (I)

Extensional view:

A (total) *function* $f: A \to B$ 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*)

What is a Function? (II)

Intensional view:

A <u>*function*</u> f: A → B is an *abstraction* λ x . e, where x is a *variable name*, and e is an *expression*, such that when a value a∈A is *substituted* for x in e, then this expression (i.e., f(a)) evaluates to some (unique) value b∈B.

What is the 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*
|	$\lambda\, x\, .\, e$	*an abstraction (function)*
|	$e_1\, e_2$	*a (function) application*

$\lambda\, x\, .\, x$ — is a function taking an argument x, and returning x

Parsing Lambda Expressions

Lambda extends as far as possible to the right

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

Application is left-associative

$$x\ y\ z \equiv (x\ y)\ z$$

Multiple lambdas may be suppressed

$$\lambda f\ g.x \equiv \lambda f\ .\ \lambda g.x$$

What is the Lambda Calculus? ...

(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:

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

β reduction substitution

Lambda expressions in Haskell

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)
```

Lambdas are anonymous functions

A lambda abstraction is just an *anonymous function.*

Consider the Haskell function:

```
compose f g x = f(g(x))
```

The *value* of `compose` is the anonymous lambda abstraction:

$$\lambda\, f\, g\, x\, .\, f\, (g\, x)$$

NB: This is the same as:

$$\lambda\, f\, .\, \lambda\, g\, .\, \lambda\, x\, .\, f\, (g\, x)$$

A Few Examples

1. $(\lambda x.x)\ y$
1. $(\lambda x.f\ x)$
2. $x\ y$
3. $(\lambda x.x)\ (\lambda x.x)$
4. $(\lambda x.x\ y)\ z$
5. $(\lambda x\ y.x)\ t\ f$
6. $(\lambda x\ y\ z.z\ x\ y)\ a\ b\ (\lambda x\ y.x)$
7. $(\lambda f\ g.f\ g)\ (\lambda x.x)\ (\lambda x.x)\ z$
8. $(\lambda x\ y.x\ y)\ y$
9. $(\lambda x\ y.x\ y)\ (\lambda x.x)\ (\lambda x.x)$
10. $(\lambda x\ y.x\ y)\ ((\lambda x.x)\ (\lambda x.x))$

Free and Bound Variables

The variable x is <u>*bound*</u> by λ in the expression: λ x.e
A variable that is not bound, is <u>*free*</u> :

$$fv(x) = \{ x \}$$
$$fv(e_1\ e_2) = fv(e_1) \cup fv(e_2)$$
$$fv(\lambda\ x\ .\ e) = fv(e) - \{ x \}$$

An expression with *no free variables* is <u>*closed*</u>.
(AKA a <u>*combinator*</u>.) Otherwise it is <u>*open*</u>.

For example, y is *bound* and x is *free* in the (open) expression:
λ y . x y

"Hello World" in the Lambda Calculus

hello world

✎ *Is this expression open? Closed?*

Why macro expansion is wrong

Syntactic substitution will not work:

$(\lambda x\, y\, .\, x\, y\,)\, y \quad \rightarrow [\, y\, /\, x]\, (\lambda\, y\, .\, x\, y)$ *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:*

$$[e/x]\, x = e$$

$$[e/x]\, y = y \qquad \textit{if } x \neq y$$

$$[e/x]\,(e_1\, e_2) = ([e/x]\, e_1)\,([e/x]\, e_2)$$

$$[e/x]\,(\lambda\, x\, .\, e_1) = (\lambda\, x\, .\, e_1)$$

$$[e/x]\,(\lambda\, y\, .\, e_1) = (\lambda\, y\, .\, [e/x]\, e_1) \qquad \textit{if } x \neq y \textit{ and } y \notin fv(e)$$

$$[e/x]\,(\lambda\, y\, .\, e_1) = (\lambda\, z\, .\, [e/x]\,[z/y]\, e_1) \qquad \textit{if } x \neq y \textit{ and } z \notin fv(e) \cup fv(e_1)$$

Consider:

$$(\lambda\, x\, .\, ((\lambda\, y\, .\, x)\,(\lambda\, x\, .\, x))\, x)\, y \rightarrow [y\, /\, x]\,((\lambda\, y\, .\, x)\,(\lambda\, x\, .\, x))\, x$$

$$= ((\lambda\, z\, .\, y)\,(\lambda\, x\, .\, x))\, y$$

Alpha Conversion

Alpha conversions allow us 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:

(λ x y . x y) y	→	(λ x z . x z) y	*α conversion*
	→	[y / x] (λ z . x z)	*β reduction*
	→	(λ z . y z)	
	=	y	*η reduction*

Eta Reduction

Eta reductions allow one to remove “redundant lambdas”.

Suppose that f is a *closed expression*
(i.e., there are no free variables in f).

Then:

$$(\lambda x . f\ x)\ y \quad \to f\ y \qquad \textit{β reduction}$$

So, (λ x . f x) behaves the same as f !

Eta reduction says, *whenever x does not occur free in f*, we can rewrite (λ x . f x) as f.

Normal Forms

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

Not all lambda expressions have normal forms!

$$
\begin{aligned}
\Omega = (\lambda x . x\, x)(\lambda x . x\, x) \rightarrow\ & [(\lambda x . x\, x) / x](x\, x) \\
=\ & (\lambda x . x\, x)(\lambda x . x\, x) && \beta \text{ reduction} \\
\rightarrow\ & (\lambda x . x\, x)(\lambda x . x\, x) && \beta \text{ reduction} \\
\rightarrow\ & (\lambda x . x\, x)(\lambda x . x\, x) && \beta \text{ reduction} \\
\rightarrow\ & \ldots
\end{aligned}
$$

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.

Non-termination

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*
$\rightarrow (\lambda x . y) ((\lambda x . x x) (\lambda x . x x))$	$\rightarrow y$
$\rightarrow (\lambda x . y) ((\lambda x . x x) (\lambda x . x x))$	
$\rightarrow \ldots$	

Compare to the Haskell expression:

(\x -> \y -> x) 1 (5/0) ⇨ 1

Currying

Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modelled as *Curried* higher-order functions.

As we have seen, to improve readability, *multiple lambdas are suppressed*, so:

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

$$\lambda\, b\, x\, y\, .\, b\, x\, y = \lambda\, b\, .\, \lambda\, x\, .\, \lambda\, y\, .\, (\, b\, x\,)\, y$$

Representing Booleans

Many programming concepts can be directly expressed in the lambda calculus. *Let us define:*

$$\text{True} \equiv \lambda\, x\, y\, .\, x$$
$$\text{False} \equiv \lambda\, x\, y\, .\, y$$
$$\text{not} \equiv \lambda\, b\, .\, b\ \text{False}\ \text{True}$$
$$\text{if } b \text{ then } x \text{ else } y \equiv \lambda\, b\, x\, y\, .\, b\, x\, y$$

then:

$$\begin{aligned}
\text{not True} &= (\lambda\, b\, .\, b\ \text{False True})\,(\lambda\, x\, y\, .\, x) \\
&\rightarrow (\lambda\, x\, y\, .\, x)\ \text{False True} \\
&\rightarrow \text{False}
\end{aligned}$$

$$\begin{aligned}
\text{if True then } x \text{ else } y &= (\lambda\, b\, x\, y\, .\, b\, x\, y)\,(\lambda\, x\, y\, .\, x)\, x\, y \\
&\rightarrow (\lambda\, x\, y\, .\, x)\, x\, y \\
&\rightarrow x
\end{aligned}$$

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:

$$\text{pair} \equiv (\lambda\, x\, y\, z\, .\, z\, x\, y)$$
$$\text{first} \equiv (\lambda\, p\, .\, p\ \text{True}\,)$$
$$\text{second} \equiv (\lambda\, p\, .\, p\ \text{False}\,)$$

then:

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

Tuples as functions

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:

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

$$succ \equiv (\lambda n . (False, n))$$

then:

$$1 \equiv succ\ 0 \rightarrow (False, 0)$$

$$2 \equiv succ\ 1 \rightarrow (False, 1)$$

$$3 \equiv succ\ 2 \rightarrow (False, 2)$$

$$4 \equiv succ\ 3 \rightarrow (False, 3)$$

...

Working with numbers

We can define simple functions to work with our numbers.

Consider:

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

then:

$$\text{iszero } 1 = \text{first (False, 0)} \quad \rightarrow \text{False}$$
$$\text{iszero } 0 = (\lambda\, p\,.\, p \text{ True})\,(\lambda\, x\,.\, x) \quad \rightarrow \text{True}$$
$$\text{pred } 1 = \text{second (False, 0)} \quad \rightarrow 0$$

✎ *What happens when we apply pred 0? What does this mean?*

What you should know!

- ✎ *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?*
- ✎ *What are normal and applicative order evaluation?*
- ✎ *Why is normal order evaluation called lazy?*
- ✎ *How can Booleans, tuples and numbers be represented in the lambda calculus?*

Can you answer these 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?*

8. Fixed Points

Overview

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

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.* I.e, plus is free in the "definition"!

Recursive functions as fixed points

We can obtain a *closed expression* by *abstracting* over plus:

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

rplus takes as its *argument* the actual plus function to use and returns as its result a definition of that function in terms of itself. In other words, if fplus is the function we want, then:

rplus fplus ↔ fplus

I.e., we are searching for a *fixed point* of rplus ...

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

Theorem:
Every lambda expression e has a *fixed point* p such that (e p) $\leftrightarrow$ p.

Proof: Let:

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

Now consider:

$$p \equiv Y e \rightarrow (\lambda x . e (x x)) (\lambda x . e (x x))$$
$$\rightarrow e ((\lambda x . e (x x)) (\lambda x . e (x x)))$$
$$= e p$$

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

How does Y work?

Recall the non-terminating expression

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

Ω loops endlessly *without doing any productive work*.

Note that (x x) represents the body of the "loop".

We simply define Y to take an *extra parameter f*, and *put it into the loop*, passing it the body as an argument:

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

So Y just inserts some productive work into the body of Ω

Using the Y Combinator

Consider:

$$f \equiv \lambda\, x.\ True$$

then:

$$Y\, f \rightarrow f\, (Y\, f) \qquad \textit{by FP theorem}$$
$$= (\lambda\, x.\ True)\, (Y\, f)$$
$$\rightarrow True$$

Consider:

$$Y\, succ \rightarrow succ\, (Y\, succ) \qquad \textit{by FP theorem}$$
$$\rightarrow (False, (Y\, succ))$$

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

Recursive Functions are Fixed Points

We seek a fixed point of:

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

By the Fixed Point Theorem, we simply take:

$$\text{plus} \equiv \text{Y rplus}$$

Since this guarantees that:

$$\text{rplus plus} \leftrightarrow \text{plus}$$

as desired!

Unfolding Recursive Lambda Expressions

plus 1 1 = (Y rplus) 1 1
→ rplus plus 1 1 *(NB: fp theorem)*
→ iszero 1 1 (plus (pred 1) (succ 1))
→ False 1 (plus (pred 1) (succ 1))
→ plus (pred 1) (succ 1)
→ rplus plus (pred 1) (succ 1)
→ iszero (pred 1) (succ 1)
(plus (pred (pred 1)) (succ (succ 1)))
→ iszero 0 (succ 1) (...)
→ True (succ 1) (...)
→ succ 1
→ 2

The Typed Lambda Calculus

There are many variants of the lambda calculus.

The *typed lambda calculus* just decorates terms with *type annotations:*

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

Operational Semantics:

$$\lambda\, x^{\tau 2} . e^{\tau 1} \Leftrightarrow \lambda\, y^{\tau 2} . [\, y^{\tau 2} / x^{\tau 2}\,]\, e^{\tau 1} \qquad y^{\tau 2} \text{ not free in } e^{\tau 1}$$

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

$$\lambda\, x^{\tau 2} . (e^{\tau 1}\, x^{\tau 2}) \Rightarrow e^{\tau 1} \qquad x^{\tau 2} \text{ not free in } e^{\tau 1}$$

Example:

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

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:

$$\mathrm{True} \equiv (\, \lambda\, x^{\alpha}\, .\, (\, \lambda\, y^{\beta}\, .\, x^{\alpha}\,)^{\beta\rightarrow\alpha})^{\alpha\rightarrow(\beta\rightarrow\alpha)}$$

$$\mathrm{True}^{\alpha\rightarrow(\beta\rightarrow\alpha)}\, a^{A}\, b^{B} \rightarrow (\, \lambda\, y^{\beta}\, .\, a^{A}\,)^{\beta\rightarrow A}\, b^{B}$$

$$\rightarrow\; a^{A}$$

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:

```
doubleLen len len' xs ys = (len xs) + (len' ys)
```

then

```
doubleLen length length "aaa" [1,2,3]
```

is ok, but if

```
doubleLen' len xs ys = (len xs) + (len ys)
```

then

```
doubleLen' length "aaa" [1,2,3]
```

is a type error since the argument `len` 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* (λ x . x x)*?*

Other Calculi

Many calculi have been developed to study the semantics of programming languages.

Object calculi: model *inheritance* and *subtyping* ..

☞ lambda calculi with records

Process calculi: model *concurrency* and *communication*

☞ CSP, CCS, π calculus, CHAM, blue calculus

Distributed calculi: model *location* and *failure*

☞ ambients, join calculus

What you should know!

- *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 these questions?

- *Are there more fixed-point operators other than Y?*
- *How can you be sure that unfolding a recursive expression will terminate?*
- *Would a process calculus be Church-Rosser?*

9. 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 with implementors as well as with 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

...

Methods for Specifying Semantics ...

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	::=	* | /

Concrete syntax is needed for parsing; abstract syntax suffices for semantic specifications.

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.

Calculator Semantics

We need three semantic functions: one for *programs*, one for *statements* (expression sequences) and one for *expressions*.

The meaning of a program is the list of integers printed:

Programs:

$$\mathbf{P} : \mathit{Program} \rightarrow \mathit{Int}^{\,*}$$

$$\mathbf{P}\,[\![\ \mathrm{ON}\ S\ ]\!] \;=\; \mathbf{S}\,[\![\ S\ ]\!]\,(0)$$

A statement may use and update LASTANSWER:

Statements:

$$\mathbf{S} :: \mathit{ExprSequence} \rightarrow \mathit{Int} \rightarrow \mathit{Int}^{\,*}$$

$$\mathbf{S}\,[\![\ E\ \mathrm{TOTAL}\ S\ ]\!]\,(n) \;=\; \mathit{let}\ n' = \mathbf{E}\,[\![\ E\ ]\!]\,(n)\ \mathit{in}\ \mathrm{cons}(n', \mathbf{S}\,[\![\ S\ ]\!]\,(n'))$$

$$\mathbf{S}\,[\![\ E\ \mathrm{TOTAL}\ \mathrm{OFF}\ ]\!]\,(n) \;=\; [\ \mathbf{E}\,[\![\ E\ ]\!]\,(n)\ ]$$

Calculator Semantics...

Expressions:

$$\mathbf{E} : \textit{Expression} \rightarrow \textit{Int} \rightarrow \textit{Int}$$

$$\mathbf{E}\,[\![\ E1 + E2\ ]\!]\,(n) = \mathbf{E}\,[\![\ E1\ ]\!]\,(n) + \mathbf{E}\,[\![\ E2\ ]\!]\,(n)$$

$$\mathbf{E}\,[\![\ E1 * E2\ ]\!]\,(n) = \mathbf{E}\,[\![\ E1\ ]\!]\,(n) \times \mathbf{E}\,[\![\ E2\ ]\!]\,(n)$$

$$\mathbf{E}\,[\![\ \texttt{IF}\ E1\ ,\ E2\ ,\ E3\ ]\!]\,(n) = \begin{array}{l} \textit{if}\ \mathbf{E}\,[\![\ E1\ ]\!]\,(n) = 0 \\ \textit{then}\ \mathbf{E}\,[\![\ E2\ ]\!]\,(n) \\ \textit{else}\ \mathbf{E}\,[\![\ E3\ ]\!]\,(n) \end{array}$$

$$\mathbf{E}\,[\![\ \texttt{LASTANSWER}\ ]\!]\,(n) = n$$

$$\mathbf{E}\,[\![\ (\ E\ )\ ]\!]\,(n) = \mathbf{E}\,[\![\ E\ ]\!]\,(n)$$

$$\mathbf{E}\,[\![\ N\ ]\!]\,(n) = N$$

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
```

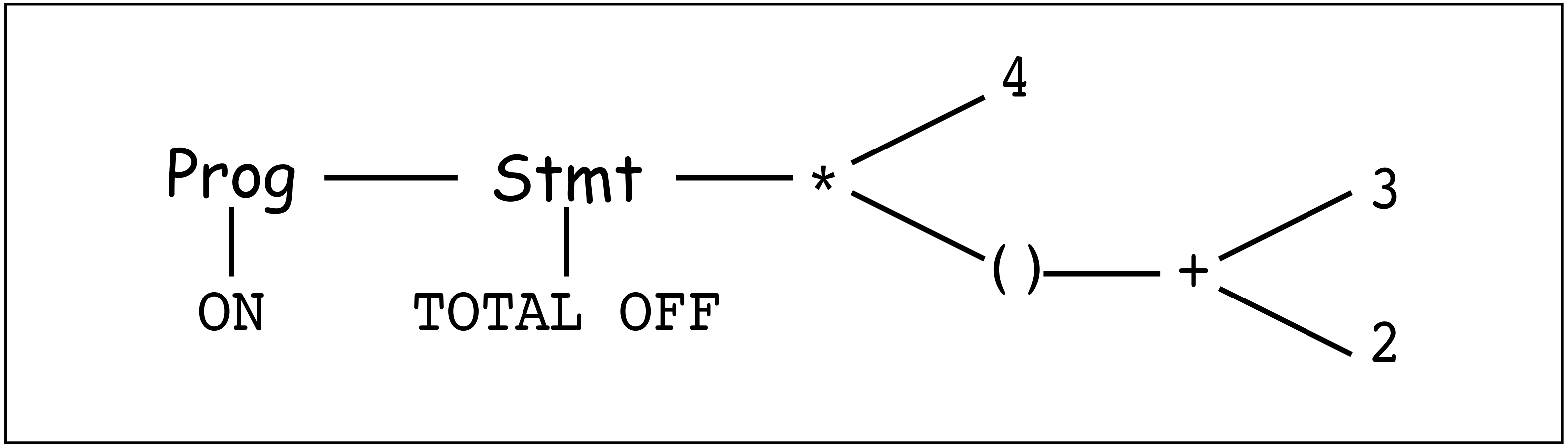
Data Structures for Abstract Syntax

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))))))
```

Implementing the Calculator

We can implement our denotational semantics directly in a functional language like Haskell:

Programs:

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

Statements:

```
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) : [ ]
```

...

Implementing the Calculator ...

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

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 .`"

Input number initializes a; output is final value of z.

Representing 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
```

An abstract syntax tree

Example:

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

Is represented as:

```
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

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

Functional updates

Example:

```
env1 = update 'a' 1 (update 'b' 2 (newstore))
env2 = update 'b' 3 env1
```

```
env1 ‘b’
➪ 2
env2 ‘b’
➪ 3
env2 ‘z’
➪ 0
```

Semantics of assignments

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

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)
```

...

Semantics of assignments ...

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

Running the interpreter

```
src1 = "z := 1 ; if a = 0 then z := 3 else z := z + a ."
ast1 = Dot (CSeq
           (Assign 'z' (Num 1))
             (If (Equal (Id 'a') (Num 0))
               (Assign 'z' (Num 3))
               (Assign 'z' (Plus (Id 'z') (Id 'a')))))
```

pp ast1 10

⇨ 11

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 ...

What you should know!

- 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 these 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?*

10. 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

What is 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
```

Prolog Questions

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 without a 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 <u>*Prolog database*</u> 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).
```

Simple queries

```
?- consult('royal').
➪ yes

?- male(charles).
➪ yes

?- male(anne).
➪ no

?- male(mickey).
➪ no
```

Just another query which succeeds

...

Queries with variables

You may accept or reject unified variables:

```
?- parent(charles, P).
➪ P = elizabeth <carriage return>
  yes
```

You may reject a binding to search for others:

```
?- male(X).
➪ X = andrew ;
  X = charles <carriage return>
  yes
```

Use anonymous variables if you don't care:

```
?- 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
```

...

Unification ...

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
```

Closed World Assumption

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

```
?- mother(elizabeth,M).
```
➪ no

```
?- male(mickey).
```
➪ 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(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) ? ...
```

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
```

Comparison ...

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 ("or") can also be expressed using the ";" operator:

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

Note that *same information* can be represented in *different* 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).

?- trace(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
```

✎ *Will ancestor/2 always terminate?*

Recursion ...

```
?- trace(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
```

✎ *What happens if you query ancestor(harry, harry)?*

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).

?- trace(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}
```

Failure

Searching can be controlled by *explicit failure:*

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

?- printall(brother(_,_)).
➪ brother(andrew,charles)
  brother(andrew,edward)
  brother(anne,andrew)
  brother(anne,charles)
  brother(anne,edward)
  brother(charles,andrew)
...
```

Cuts

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

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

Cut says to Prolog:

> *"This is the right answer to this query. If later you are forced to backtrack, please do not consider any alternatives to this decision."*

Negation as failure

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

```
not(X) :- X, !, fail.     % if X succeeds, we fail
not(_).                   % if X fails, we succeed
```

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(_,_).
```

Changing the Database ...

```
?- male(charles); parent(charles, _).
⇨ yes
?- rename(charles, mickey).
⇨ yes
?- male(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.
⇨ X = 11 ?
```

Is *syntactic sugar* for:

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

Defining Functions

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.

?- fact(10,F).
➪ F = 3628800 ?
```

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 , b)	[a | b]	[a | b]
.(a , .(b , c))	[a | [b | c]]	[a , b | c]

Lists can be *deconstructed* using cons pair syntax:

```
?- [a,b,c] = [a|X].
```

⇨ `X = [b,c]?`

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
```

Pattern Matching with Lists ...

Prolog will automatically *introduce new variables* to represent unknown terms:

```
?- in(a, L).
⇨ L = [ a | _A ] ? ;
  L = [ _A , a | _B ] ? ;
  L = [ _A , _B , a | _C ] ? ;
  L = [ _A , _B , _C , a | _D ] ?
  yes
```

Inverse relations

A carefully designed relation can be used in many directions:

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

?- append([a],[b],X).
➪ X = [a,b]

?- append(X,Y,[a,b]).
➪ X = [] Y = [a,b] ;
  X = [a] Y = [b] ;
  X = [a,b] Y = []
yes
```

Exhaustive Searching

Searching for permutations:

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

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

Limits of declarative programming

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]).
```

Of course, efficient solutions in Prolog do exist!

What you should know!

- ✎ 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 these 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?*

11. Applications of Logic Programming

Overview

- I. *Search problems*
  - SEND + MORE = MONEY
- II. *Symbolic Interpretation*
  - Definite Clause Grammars
  - Interpretation as Proof
  - An interpreter for the calculator language

Reference

- The Ciao Prolog System Reference Manual, Technical Report CLIP 3/97.1, www.clip.dia.fi.upm.es

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).
```

A non-solution ...

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

But this doesn't work because "is" can only evaluate expressions over *instantiated variables.*

```
?- 5 is 1 + X.
➪» evaluation_error: [goal(5 is
  1+_64),argument_index(2)]
  [Execution aborted]
```

A first solution

So let's instantiate them first:

```
digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).
digits([]).
digits([D|L]):- digit(D), digits(L).

% pick arbitrary digits:
soln1 :- digits([S,E,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,     % check if solution is found
         showAnswer(A,B,C).
```

A first solution ...

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).
```

A second (non-)solution ...

Maybe it works. We'll never know ...

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

after 8 minutes still running ...

✎ *What went wrong?*

A third solution

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

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

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

?- carrysum([5,6,7],D,C).
➪ D = 8
  C = 1
```

A third solution ...

We instantiate the final digits first, and use the carrysum to *constrain the search space:*

```
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,
         showAnswer(A,B,C).
```

A third solution ...

This is also correct, but uninteresting:

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

A fourth solution

Let's try to make the variables *unique:*

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

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

?- unique([a,b,c]).
⇨ yes
?- unique([a,b,a]).
⇨ no
```

A fourth solution ...

```
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).
```

A fourth solution ...

This works (at last), in about 1 second on a G3 Powerbook.

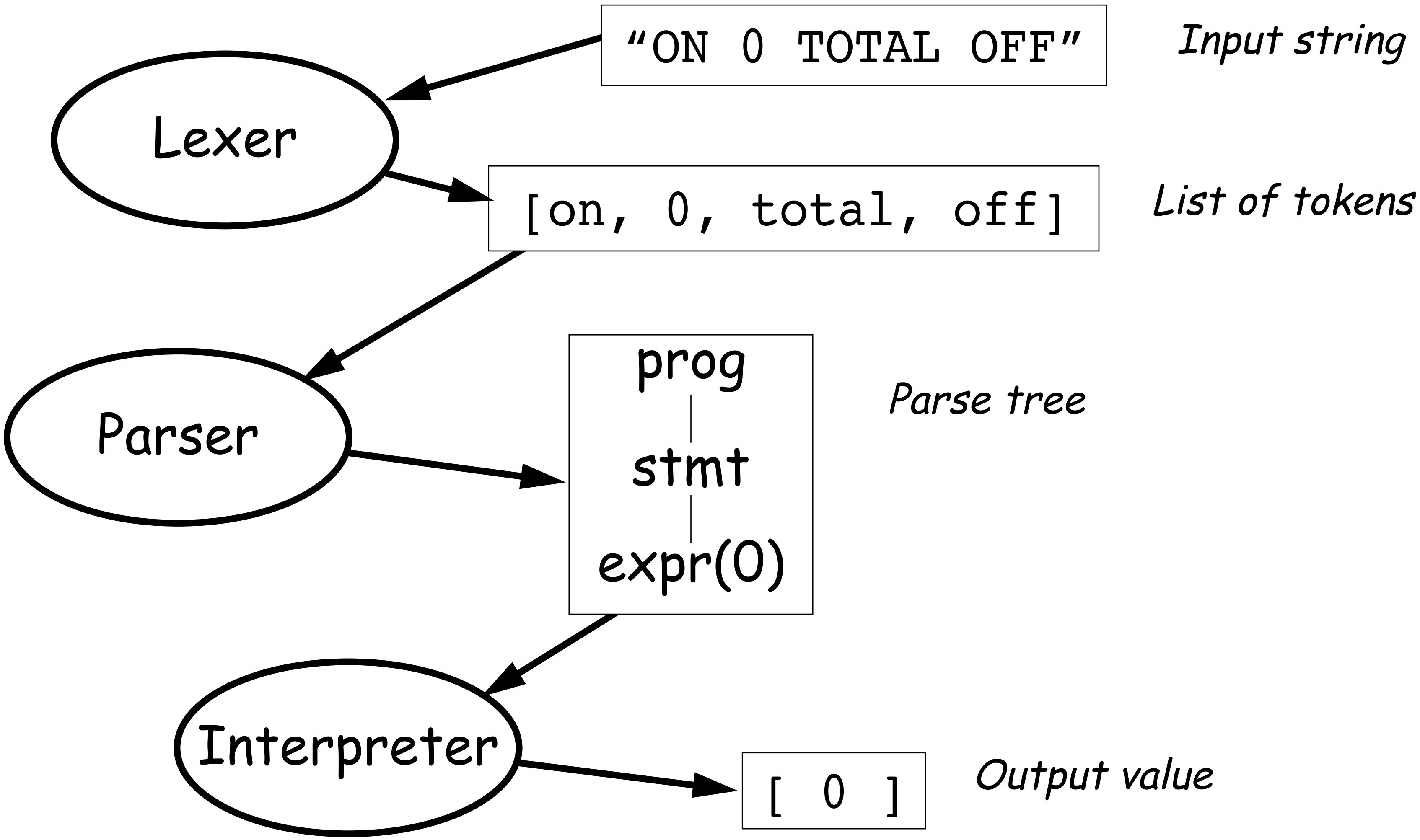
```
soln4.
➪   9567 + 1085 = 10652
    yes
```

II. Symbolic Interpretation

Prolog is an ideal language for implementing small languages:

- Implement BNF using Definite Clause Grammars
- Implement semantic rules directly as Prolog rules

Goal-directed interpretation



Definite Clause Grammars

Definite clause grammars are an extension of context-free grammars.

A DCG rule in Prolog takes the general form:

```
head --> body.
```

meaning “a possible form for head is body”.

The head specifies a non-terminal symbol, and the body specifies a sequence of terminals and non-terminals.

Definite Clause Grammars ...

- *Non-terminals* may be any Prolog *term* (other than a variable or number).
- A sequence of zero or more *terminal* symbols is written as a Prolog *list*. A sequence of ASCII characters can be written as a string.
- *Side conditions* containing Prolog goals may be written in { } brackets in the right-hand side of a grammar rule.

Example

This grammar parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {0'0=<C, C=<0'9, X is C - 0'0}.
```

How to use this?

The query

```
| ?- expr(Z, "-2+3*5+1", []).
```

will compute `Z=14`.

How does it work?

DCG rules are just syntactic sugar for normal Prolog rules.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
```

translates to:

```
expr(Z, S0, S) :-              % input and goal
  term(X, S0, S1),             % pass along state
  'C'(S1,43,S2),               % "+" = [43]
  expr(Y, S2, S),
  Z is X + Y .
```

'C' is a built-in predicate to recognize terminals.

Lexical analysis

We can use DCGs for both scanning and parsing.

Our lexer will convert an input atom into a list of tokens:

```
lex(Atom, Tokens) :-
  name(Atom, String),
  scan(Tokens, String, []), !.

scan([T|Tokens]) -->
              whitespace0, token(T), scan(Tokens).
scan([]) --> whitespace0.
```

Recognizing Tokens

We will represent simple tokens by Prolog atoms:

```
token(on)    --> "ON".
token(total) --> "TOTAL".
token(off)   --> "OFF".
token(if)    --> "IF".
token(last)  --> "LASTANSWER".
token(',')   --> ",".
token('+')   --> "+".
token('*')   --> "*".
token('(')   --> "(".
token(')')   --> ")".
```

Recognizing Numbers

and a number N by the term num(N):

```
token(num(N))  --> digits(DL), { asnum(DL, N, 0) }.

digits([D|L])  --> digit(D), digits(L).
digits([D])    --> digit(D).

digit(D)       --> [D], { "0" =< D, D =< "9" }.
```

✎ *How would you implement asnum/3?*

Concrete Grammar

To parse a language, we need an unambiguous grammar!

```
p     ::=   'ON' s
s     ::=   e 'TOTAL' s
       |    e 'TOTAL' 'OFF'
e     ::=   'IF' e1 ',' e1 ',' e1
       |    e1
e1    ::=   e2 '+' e1
       |    e2
e2    ::=   e3 '*' e2
       |    e3
e3    ::=   'LASTANSWER'
       |    num
       |    '(' e0 ')'
```

Parsing with DCGs

The concrete grammar is easily written as a DCG:

```
prog(S)                   --> [on], stmt(S).
stmt([E|S])               --> expr(E), [total], stmt(S).
stmt([E])                 --> expr(E), [total, off].
expr(E)                   --> e0(E).
e0(if(Bool, Then, Else)) --> [if], e1(Bool), [','],
                                    e1(Then), [','], e1(Else).
e0(E)                     --> e1(E).
e1(plus(E1,E2))           --> e2(E1), ['+'], e1(E2).
e1(E)                     --> e2(E).
e2(times(E1,E2))          --> e3(E1), ['*'], e2(E2).
e2(E)                     --> e3(E).
e3(last)                  --> [last].
e3(num(N))                --> [num(N)].
e3(E)                     --> ['('], e0(E), [')'].
```

Representing Programs as Parse Trees

We have chosen to represent *expressions* as Prolog *terms*, and *programs* and statements as *lists* of terms:

```
parse(Atom, Tree) :-
    lex(Atom, Tokens),
    prog(Tree, Tokens, []).

parse(
    'ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF',
    [ times(plus(num(1),num(2)),
            plus(num(3),num(4))),
      plus(last,num(10))
    ])
```

Testing

We exercise our parser with various test cases:

```
check(Goal) :- Goal, !.
check(Goal) :-
  write('TEST FAILED: '),
  write(Goal), nl.

parseTests :-
  check(parse('ON 0 TOTAL OFF', [num(0)])),
  ...
```

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

Building a Simple Interpreter

We define semantic predicates over the syntactic elements of our calculator language.

```
peval(S,L)                   :-  seval(S, 0, L).

seval([E], Prev, [Val])      :-  xeval(E, Prev, Val).
seval([E|S], Prev, [Val|L])  :-  xeval(E, Prev, Val),
                                 seval(S, Val, L).

xeval(num(N), _, N).
xeval(plus(E1,E2), Prev, V)  :-  xeval(E1, Prev, V1),
                                 xeval(E2, Prev, V2),
                                 V is V1+V2.
```

...

Running the Interpreter

The interpreter puts the parts together

```
eval(Expr, Val) :-
  parse(Expr, Tree),
  peval(Tree, Val).

eval(
    'ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF',
    X).
```

⇨ `X = [21, 31]`

Testing the interpreter

We similarly define tests for the interpreter.

```
evalTests :-
    check(eval('ON 0 TOTAL OFF', [0])),
    check(eval('ON 5 + 7 TOTAL OFF', [12])),
    ...
```

A top-level script

Finally, we can package the interpreter as a ciao module, and invoke it from a script:

```
#!/bin/sh
exec ciao-shell $0 "$@" # -*- mode: ciao; -*-
:- use_module(calc, [eval/2, test/0]).
main([])    :- test.
main(Argv) :- doForEach(Argv).
doForEach([]).
doForEach([Arg|Args]) :-
  write(Arg), nl,
  eval(Arg, Val),
  write(Val), nl,
  doForEach(Args).
```

What you should know!

- What are *definite clause grammars*?
- How are DCG specifications *translated* to Prolog?
- Why are *abstract grammars* inappropriate for parsing?
- Why are *left-associative* grammar rules problematic?
- How can we represent *syntax trees* in Prolog?

Can you answer these 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?
- ✎ Why must DCG side conditions be put in { curly brackets }?
- ✎ What exactly does the 'C' predicate do?
- ✎ Why do we need a separate lexer?
- ✎ How would you implement an interpreter for the assignment language we defined earlier?

12. Piccola — A Small Composition Language

Handouts will be distributed before the lecture.

13. Summary, Trends, Research ...

- Summary: functional, logic and object-oriented languages
- Research: ...
  - ☞ www.iam.unibe.ch/~scg

C and C++

Good for:

- systems programming
- portability

Bad for:

- learning (very steep learning curve)
- rapid application development
- maintenance

Trends:

- increased standardization
- generative programming

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:

- object calculi
- concurrent, distributed calculi (e.g., π calculus, “join” calculus ...)

Type Systems

Good for:

- catching static errors
- documenting interfaces
- formalizing and reasoning about domains of functions and objects

Bad for:

- reflection; 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
- modules

Object-Oriented Languages

Good for:

- domain modelling
- developing reusable frameworks

Bad for:

- learning (steep learning curve)
- understanding (hard to keep systems well-structured)
- semantics (no agreement)

Trends:

- component-based software development
- aspect-oriented programming

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