Flavour of Languages

Ashish Mahabal
aam@astro.caltech.edu
Caltech, 7 Apr 2011
Ay199/Bi 199b
Quick survey

- C?
- Shell?
- Perl?
- Python?
- HTML?
- SQL?
The language you use influences how you think (about problems)

- Types of languages
- Features of languages
- Internal issues
- Extendibility, domain specific languages
- Available help, practical issues
- Architecture/compilation etc.
- Wider issues(?)
- Exercise
How to shoot yourself in the foot
(http://www-users.cs.york.ac.uk/~susan/joke/foot.htm)

• C: You shoot yourself in the foot
• C++: You accidently create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me over there."
• FORTRAN: You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception handling ability.
If languages were religions
(http://www.aegisub.net/2008/12/if-programming-languages-were-religions.html)

• C would be Judaism - it's old and restrictive, but most of the world is familiar with its laws and respects them. …

• C++ would be Islam - It takes C and not only keeps all its laws, but adds a very complex new set of laws on top of it. …

• Lisp would be Zen Buddhism

• Perl would be Voodoo

• Python would be Humanism

• …. 
Types of languages
(its difficult to put a single label actually)

• Imperative (e.g. C, Java, …)
• Functional (e.g. LISP, Haskell, perl, python, …)
• Logical (e.g. prolog)
• ...
• Formatting/markup (e.g. HTML, XML, KML, …)
• ...
• Database (e.g. SQL and its flavours)
• ...
• Shells (e.g. tcsh, bash, ksh, …)
logic programming contributes **non-determinism, inversion and partial data structures**, whereas functional programming contributes **efficient evaluation and infinite data structures**.

(http://web.cecs.pdx.edu/~antoy/research/flp/index.html)
Imperative(C, Java)

- Computation as statements that change program state
  - i = 0;
  - i++;
    - n=10;
  - j=1;
  - for(i=2;i<=n;++i){j*=i;}

Procedural (perl, python)

• Method of executing imperative language programs (imperative + subprograms)

```perl
sub fact_rec {  # recursive
  my $n = shift;

  return undef if $n < 0;
  return 1  if $n <= 1;
  return $n * fact_rec( $n-1 );
}
```

(Could have issues in list mode).
Functional (Haskell, LISP)

• computation as the evaluation of mathematical functions. No state.
• Effected through lambda calculus, composition of functions

```haskell
let rec fact = lambda n. if n=0 then 1 else n*fact(n-1) in fact 10
```
Logical (Prolog)

• Define “what” is to be computed rather than “how” (declarative: properties of correct answers)

```prolog
factorial(0,1).
factorial(A,B) :-
    A > 0,
    C is A-1,
    factorial(C,D),
    B is A*D.
?- factorial(10,What).
What=3628800
```
markup/database

- SGML/HTML/XML – stylized rendering (XML to be covered in other talks)
  - Tags used for formatting
  - `<A HREF=SomeLink>SomeText</A>`
  - `<mytag>lalala</mytag>`
- KML – Keyhole Markup Language
  - Convert points for Google Earth/sky locations
- SQLs e.g. my, ms, pg, ... (SQL and databases will also be covered in detail in other talks)
  - For talking to databases
  - `Select * from TableX where Y=Z`
shells

• bsh/bash/csh/ksh/tcsh ... are languages in their own right
  – awk/sed/grep
  – History
    “!mv; !scp:p; ^my^ny”
  – Loops
    • foreach f (*.jpg)
    • convert $f $f:r.png
    • end
  – Redirections
    “(myprog < myin > myout) >& myerr &”
  – Scripts
    “at now + 24 hours < foo.csh”
For Matlab buffs

Optimization

The Computer Language Benchmarks Game

http://shootout.alioth.debian.org

Benchmarking programming languages?
How can we benchmark a programming language?
We can't - we benchmark programming language implementations.

How can we benchmark language implementations?
We can't - we measure particular programs.
sub fact_rec {  # recursive
    my $n = shift;
    return undef if $n < 0;
    return 1 if $n <= 1;
    return $n * fact_rec( $n-1 );
}

sub fact_loop {  # looping
    my $n = shift;
    return undef if $n < 0;
    return 1 if $n <= 1;
    my $prod = my $k = 1;
    $prod *= ++$k while $k < $n;
    return $prod;
}

my @fact_cache = ( 1 );
sub fact_cache {  # cache results of looping
    my $n = shift;
    return undef if $n < 0;
    return $fact_cache[$n] if $n <= $#fact_cache;
    my $prod = $fact_cache[-1];
    push( @fact_cache, $prod *= $#fact_cache )
        while $#fact_cache < $n;
    return $prod;
}

And then there is built-in memoizing
Features of Languages

• strong/weak/no typing; datatypes
• safe/unsafe typing
• dynamic/static datatype conversions
• side effects/monads
• concurrency
• distributedness
strong/weak typing

• `#include <stdio.h>`
  `main()`
  `{int fill; fill=42; printf("\%s\n", fill);}`
strong/weak typing

• "#include <stdio.h>                             main()
  \{int fill; fill=42; printf("%s\n",fiil);\}"
– This will not compile for at least two reasons:
  • fiil (mistyped) is not declared
  • Even if that is corrected, it is not a string
strong/weak typing

• \texttt{#include <stdio.h>} \quad \texttt{main()}
\begin{verbatim}
{int fill; fill=42; printf("%s\n",fiil);}
\end{verbatim}

– This will not compile for at least two reasons:
• fiil (mistyped) is not declared
• Even if that is corrected, it is not a string

• \texttt{#!/usr/bin/perl}
\begin{verbatim}
$fill=42;printf("%s\n",$fiil);
\end{verbatim}
strong/weak typing

- `#include <stdio.h>`
  ```c
  main()
  {int fill; fill=42; printf("%s\n",fiil);}
  ```
  - This will not compile for at least two reasons:
    - `fiil` (mistyped) is not declared
    - Even if that is corrected, it is not a string

- `#!/usr/bin/perl`
  ```perl
  $fill=42;printf("%s\n",$fiil);
  ```
  - This also fails, but silently. No error is announced
  - Change `fiil` to `fill` (leaving it as `%s`) and you get the correct result (by coincidence)
• `#!/usr/bin/perl -w`
• `use strict;`

A language is only as rigid or flexible as your understanding of it.

Grammars: (Extended) Bachus-Nour form

```plaintext
 ::= <> " " [ ] | {}
```
Partial grammar for C

<multiplicative-expression> ::= <cast-expression>
   | <multiplicative-expression> * <cast-expression>
   | <multiplicative-expression> / <cast-expression>
   | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
   | ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>
   | ++ <unary-expression>
   | -- <unary-expression>
   | <unary-operator> <cast-expression>
   | sizeof <unary-expression>
   | sizeof <type-name>
Extendibility

• With other languages
  – Perl through C
  – C through perl

• Packages for particular domains and their extensibility (e.g. matlab/iraf/idl)
  – Domain specific core functionality
  – Can be extended further using packages
• Domain specific languages
  – Define terms/keywords close to the domain
  – Overload terms in domain appropriate way
    – select RA,Dec from PQ where mag > 15
    join radio > 1Jy
Other esoteric sounding but important stuff

• syntactic sugar
  – \( a[i] \) rather than \(*(a+i)\)
  – \( a[i][j] \) rather than \(*(*(a+i)+j)\)

• side effects/monads

Avoid the pitfall of division by 0 by returning a “maybe” monad of value “nothing”
• Lazy evaluation (delayed until needed)
  • x=f(y) will remain as is until x is needed
  • Possible to define infinite lists
  • Control structure: \( a==b?c:d \)

• Haskell’s implementation of Fibonacci numbers

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

• constant folding/argumentless functions

(evaluating constants at compile time)

```c
int f (void) {
    return 3 + 5;
}
```
Help Available

• debugging tools
  – Internal debuggers
  – External/graphical debuggers
    • `perl –c` checks syntax
    • `perl –d` default die handler
    • `ddd` debugger (works with most language debuggers)

• Macro editing modes
  – Emacs, vim (autotab, headers, brace matching)
ddd
Practical Issues

• **OS support**
  – *perl/c* supported on practically all platforms

• **ease of learning (how to shoot your foot ...)**
  – Functional/logical may seem non-intuitive initially
  – So do java and C++

• **readability across teams**
  – Structure of syntax e.g. tabs in python

• **Speed, scalability, reusability**
Wider issues

• We have scratched only the surface
  – Did not even mention entities like
    • Postscript
    • Tcl
    • Text processing
• Non-Von Neumann computers
Larry Wall in ‘State of the Onion’ (2006)

• Early/late binding
• Single/multiple dispatch
• Eager/lazy typology
• Limited/rich structures
• Symbolic/wordy
• Immutable/mutable classes
• Scopes (various kinds)
## Perligata (Damian Conway)

<table>
<thead>
<tr>
<th>Perligata</th>
<th>Number, Case, and Declension</th>
<th>Perl</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextum</td>
<td>accusative singular 2nd</td>
<td>$next</td>
<td>scalar data</td>
</tr>
<tr>
<td>nexta</td>
<td>accusative plural 2nd</td>
<td>@next</td>
<td>array data</td>
</tr>
<tr>
<td>nextus</td>
<td>accusative plural 4th</td>
<td>%next</td>
<td>hash data</td>
</tr>
<tr>
<td>nexto</td>
<td>dative singular 2nd</td>
<td>$next</td>
<td>scalar target</td>
</tr>
<tr>
<td>nextis</td>
<td>dative plural 2nd</td>
<td>@next</td>
<td>array target</td>
</tr>
<tr>
<td>nextibus</td>
<td>dative plural 4th</td>
<td>%next</td>
<td>hash target</td>
</tr>
<tr>
<td>nexti</td>
<td>genitive singular 2nd</td>
<td>[$next]</td>
<td>indexed scalar</td>
</tr>
<tr>
<td>nextorum</td>
<td>genitive plural 2nd</td>
<td>$next[]</td>
<td>indexed array</td>
</tr>
<tr>
<td>nextuum</td>
<td>genitive plural 4th</td>
<td>$next{}</td>
<td>indexed hash</td>
</tr>
</tbody>
</table>
Von Neumann architecture

- instructions and data are distinguished only implicitly through usage
- memory is a single memory, sequentially addressed
- memory is one-dimensional
- meaning of the data is not stored with it
- Things looking better with Virtual machines and multi-core processors
Things we have left out

- Interpreters/compilers and the vagueness in between
- memory management
- garbage collection
- bytecode
- virtual machines
- Many core (parallelism)
New Programming Paradigm

- 512 cores & 16×1536 ~25k threads per GPU
  - Running a billion threads a second
- Forget the fancy old algorithms
  - Built on wrong assumptions
- Today ALU is free, RAM is slow
  - GPU has >150 GB/s bandwidth
  - Still difficult to occupy the cores
C for CUDA

- Clean and simple

```c
int main()
{
    ...
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}

// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```
Currently Available

- GPU optimized Sorting, RNG, BLAS, FFT, Hadamard...
- SDK w/examples
- Nsight debugger!
- Imaging routines
- Python w/ PyCUDA
- High-level C++ programming with Thrust
Projects on CUDA Zone

CUDA COMMUNITY SHOWCASE
View over 1000 papers and apps developed on the CUDA architecture by programmers, scientists, and researchers around the world.

...more
CUDA EVENTS
CUDA TECHNOLOGY
WHAT IS CUDA?
CUDA IN ACTION
CUDA Showcase
CUDA for Research
CUDA for Medical
CUDA for Video & Photos
CUDA for Energy
Cross-matching

- C for CUDA prototype
  - No smart I/O, RAM limit
- NVIDIA GTX 480 1.5GB
  - 5” search with 5” zones
  - 29M×29M in **11 seconds**!
Horses for courses

- Don't marry a particular language
- Know one well, but do sample many other
- Use a language close to your domain
- Use tools which aid during programming
Hamming (regular) numbers

- $2^i \times 3^j \times 5^k$ (int $i,j,k \geq 0$)
- 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, ...
- Merge these lists:
  - 1;
  - 2, 4, 8, 16, ...;
  - 3, 9, 27, 81, ...;
  - 5, 25, 125, ...
  - Is 7 in the list? 10? 333?
let rec
merge = lambda a. lambda b.
  if hd a < hd b then (hd a)::(merge tl a b)
  else if hd b < hd a then (hd b)::(merge a tl b)
  else (hd a)::(merge tl a tl b),

mul = lambda n. lambda l. (n * hd l)::(mul n tl l)

in let rec
  hamm = 1 :: (merge (mul 2 hamm)
              (merge (mul 3 hamm)
                    (mul 5 hamm)))

in hamm
Exercise

• Write a program to generate Hamming numbers in at least 3 different (types?) of languages

• Compare them against each other in a few different ways (speed, memory, typing requirements)

• Use a debugger during the exercise and when testing it

In J: \n\ \ \ hamming=: \{./:\~@~.@], 2 3 5 * \}/@ (1x,~i.@-) \nhamming 20