## Lecture 1:

 Over 1000 languages, about 25 relative widespread

 How do we convert our solution from one language to another

Writing a program has two parts:

* + Problem solving
  + Expressing in PL

Working of Compiler and Interpreters:

* + Semantics
  + Execution time
  + Memory usage
  + Programming platform

Best way to learn programming:

* + Decide on a project
  + Find an open source program that performs a similar task
  + Start hacking and modifying the open source program
  + Read up and learn on a need-to-know basis

What drives the development of PL:

Commercial Language -- Software productivity

* + Simple execution model
  + Fill a specific niche
  + Robustness - less bug
  + Modularity - easy collaboration
  + Reusability: libraries
  + Maintainability: adding new features do not require lots of changes
  + Security: mobile code can be trusted
  + Enduring paradigm: OO
  + New Driver: customer loyalty (or.. Commonly used by the programmers)

Research languages:

* + Novel ways of expressing computation
  + Complicated execution model
  + Complicated problems get simple solutions in new pradigms
  + Useful for "proof of concept" projects
  + Inherently slow
  + Elegant , puristic approach

Machine generated alternative text: Short History: Commercial Languages
. Assembly languages (ealry 1950s)
. Fortran (late 1950s)
. Algol (1960s) ? precursor of PL/1, Modula, Pascal
. Cobol (1960s)
. C (1973) ? Portable assembly language
. Ada (1970s) ? introduces concurrency
. SQL (late 1970s)
? C++ (1985) ? Borrowed COP from Smalltalk
. Java (1995)
. PerI, Python, Javascript, PHP, VB (1990s)
. C# (2000)
. Go (2009) ? Fine-grain concurrency in a compiled language

Machine generated alternative text: Short History: Research Languages
. Lisp (1958) ? LISt Processing
? Garbage i meta-circularity
? Many variants, including Scheme
. Prolog (1972) ? Programming in Logic
? Rule based programming, dynamic syntax modification,
meta-ci rcularity
. Smalltalk (1972) ? Object orientation
. ML (1970s) ? typed functional programming
? Precursor of Standard ML, Oca ml
. CLP (1980)? Constraint Logic Programming
. Haskell (1990)
? Algebraic programming, lazy functional programming
. Oz (1991)
? M ultiparadigm, fine-grained concurrency

Programing Paradigms:

* + Imperative Programming
  + Logic Programming
  + Functional Programming
  + OO Programming
  + Constraint Programming
  + Event-Driven ..
  + Aspect-Oriented ..

The Good of C language:

* + Portable assembly language
  + Low overheads compared to real assemly languages
  + Most compiliers and interpreters for other languages are written in C

Declaration, Expression, statement;

Function: function prototype, function definition,

formal argument, actual argument

Imperative paradigm:

* + Sequential execution of statements
  + Based on the notion of state
  + Each statement takes the current state to a new state

For => while:

Machine generated alternative text: mt mamo {
Alternative way of mt  . ;
computing the same  for ( suinO, j = 0 ; j < 10 ; sum+=i, j ++ ) ;
thing. } ...
mt mair?) {
mt sum, j ;
sum = j = 0 ;
while C  < ? ) { ,_/ Equivalent while ioo.
sum += j ;
j ++ .
...

Machine generated alternative text: C-by-Example: goto statements
mt power(int a, mt b) { mt power(int a, mt b) {
ints=1,cnt=31; ints=1;
(b<O) returnO ;
return s ;
Systematic translation scheme

Machine generated alternative text: Simulation of break/continue with goto
mt sumpos ( mt a
mt j, sum ;
for ( j = O,
j <n
sum -
return sum ;
A
JI
mt sumpos ( mt aC], mt n ) {
mt sumpos ( in?
mt j, sum
mt sumpos ( mt a[], mt n ) {
mt i, sum ;
for ( i = O, sum = 0 ;
i<n;
sum += a[i++] )
if ( a[i] < 0 )IbreI;
return sum ;
mt j, sum ;
for ( j = O, sum = 0 ; j < n ; i++ )
if ( a[i] < 0 )Icontinuel;
sum += a[iJ ;
}
return sum ;
}
{
tri) {
for ( j %, sum = O j < n ; j++ ) -
if ( i] < 0 ) pgoto cont I;
= a[iJ ;
}
return sum ;
}

Machine generated alternative text: mt accept(char aU
mt j = O;
void *cas
{
Computed goto statement Addresses of labels
1, L2, L3, L4
O;
.- ,
while (J < n ) {
if (state > 3) return 0 ;
莏o *caseptr[staiiT]
fiLl : // old case O
/ if (aU] == 慳?
f state= 1; gotoL5 ;
else return 0 ;
L2 : II old case 1
if (aU] == 慴?
state = 2 ; goto Lb ;
else return 0 ;
L3 : /1 old case 2
switch(a[i]) {
case 慳? : state = 1 ;
goto Lb ;
case 慴? : state = 3 ;
goto Lb ;
default : return 0 ;
}
L4 : // old case 3
switch(a[i]) {
case 慳? : state = 1 ;
goto Lb ;
case 慴? : state = 0 ;
goto Lb ;
default : return 0 ;
}
>
Lb: i ++ ;
>
if ( state == 0 ) return 1 ;
else return 0 ;
Computed goto
jumps to one of Li, [2, [3, [4
depending on state
}
Slide 16 of 25
CS2104 ? Lectu

Machine generated alternative text: Example
Equivalent VAL code
Original code:
mt power(int a, mt b) {
mt s = 1, cnt = 31 ;
if (b<O) returnO ;
while ( --cnt>=
if ( b & (1?0) )
mt eax, ebx, ecx, edx, esi, edi ;
unsigned char M[10000] ;
II arguments a,b in ecx, edx
II return value in eax, at end of f unc
void exec() {
eax 1; ebx 31 ;
if( edx < 0 ) goto return_0
.
憃p
ebx - 1 ;
s * s * a ; 
} else
( ebx < 0)
edx; ec
=s,
? 1 ?
(edi_)goto then_branch ;
Jeax ;: eaiE?-
-oto end_if
(ien_branch:
return s ;

## Lecture 3

Language:

* + Symbol: elements of an alphabet
  + String: sequences of symbol
  + Language: set of string

Specification of Language:

* + Grammar: standard mechanism of specifying programming languages

Formal def of grammar:

Machine generated alternative text: Formal Definition of Grammar
Grammar: tuple G (, N. fi, S)
- alphabet of terminal symbols
N - set of non-terminal symbols
fi - set of production rules
S - start non-terminal, S C N
C(G) C(S) (contains only strings of terminal symbols)

Machine generated alternative text: Backus-Naur Form
Original grammar:
E?(E)
E - E-E
E ? EE
E桪
D -* UD Equivalent BNF:
D-*ID
D ? 2D
D 梖 3D expr) ::= ?? (expr) ??
D ? 4D
D , SD
D 6D (expr) [?? I? * 搄 (expr)
D ? W _____
D 梖
D ? 9D
?I???扞?扞?扞??????
D?O
D?1
D?2
D One or more repetitions
D ? s I Iteration I
D?6 _________________
D ? IFor O or more repetitions, use * I

Parsing: building an AST for a given string, provided it's in the grammar's language

Machine generated alternative text: Regular Languages
A right regular grammar (also called right linear grammar) is a
grammar (N, Z, P, S) such that all the production rules in P
are one of the following forms:
(a) B??,whereBeNandaeZ.
(b) B?aC, whereB,CeNand aeZ.
(c) B ? E, where B e N and E is the empty string.
A similar definition exists for left regular grammars. Given a
left regular grammar GL, there exists a right regular grammar
GR such that ?GL) =

Machine generated alternative text: Regular Expressions
? Language that defines languages (like the BNF)
? Equivalent to regular grammars and DFAs
? For every RE E, there exists an RG G and a DFA A
s.t.C(E) = ?G) = C(A). The converse is also true.
Definition: Let  = {a, b, c... .} be an alphabet of
sym bols.
? c is an RE with C(c) = {c}
? For each x  , x is an RE with C(x) = {x}
? Given two REs r and s, rs is an RE with
C(rls) = C(r) U C(s).
? Given two REs r and s, rs is an RE with
C(rs) = C(r) ? C(s).
? Given an RE r, r* is an RE with C(r*) = U>o(C(r)).

Machine generated alternative text: Rubys RE usage
Syntax: / ... /
First-class value: x = /ab*c/
RE matching: x.match 搇abbc2?
Evaluates to: ?tabbc?

Machine generated alternative text: Unification Algorithm for Terms with Vars
(O) Initial unification request: > = lii. > = 112.... ; answer is empty
(1) If > is a variable, add  = 11 to the answer, and replace  by 11 everywhere in the query
and rhs of bindings in answer. Remove > = 11 from current unification request, and go to
last step.
(2) If I1 is a variable, add 11 = > to the answer, and replace I1 by  everywhere in the query
and rhs of bindings in answer. Remove  = 11 from current unification request, and go to
last step.
(3) If functor(>1) functor(11i) or arity(1) # arity(111) stop with failure.
(4) If arity(D1) = O, remove >. = 11 from current unification request and go to last step.
(5) Denote by  12 ... -1k the arguments of j, and by 揻l? 1112.. .. ?k the arguments
of Il?
(6) Set the new unification request to
= 1111e >.12 = 1112.. . . >1k = ?k 2 = 112....
(7) If current unification request is not empty, go to the first step, otherwise terminate with
success.

Machine generated alternative text: How op Works?
o ?- op(Precedence,Associativity,Symbol).
o Precedence: 1-1200, lower value binds more tightly
o Associa tivity:
? yfx : binary with left associativity
? xf y : binary with right associativity
? xfx : binary with no associativity
? fx : unary prefix, non-associative
? fy : unary prefix, associative
? xf : unary postfix, non-associative
? xy : u nary postfix, associative
o Symbol: the new operator, can be any atom.

Machine generated alternative text: Lists
o Lists are widely used in Prolog
o The list of 3 elements a, b, C:
? [a,b,c]
? syntactic sugar for . (a, . (b, . (c, [])))
? alternative 慶ons? writing: [al [bI [cl []]]]
o [HIT] . (H,L) = the list with head H and tail T.
o Lists are just regular terms
? any other functor could be used to encode sequences

Function Pointer:

Machine generated alternative text: Hierarchic Data
).
return & g ;
mt ain() {
mt (*(*.a) [3J) [10]
a . (mt (*(*.)[))[1OJ)malloc(sizeof(mnt (*(.)[])[10])) ;
? (mt (*(.)[])[iO])rnalloc(sizeof(int (.[3])[10))) ;
. (mt (*(.)[])[10J)a11oc(sizeof(mnt (*(.)[])[1O])) ;
g

Pointer:

Machine generated alternative text: mt main
mt ( ()[3])[iO] ;
a\(鐃 (*(s*)[iiO])aia11oc(sizeo?mnt (*(*)[])[1O]y;
*a \(? (*(*) [j) r?])rna11oc(sizeo?int (*[3]) [1O])1f;
*(a+1K\nt (s(*)[Thk[1O])malloc(sizeof(mnt (*(.)5[iO])) ;
(**a)[O\ ?iit (*)[iOra11oc(siz€of(mnt [1O)))罀
(*(**a) [(
(*鞶) (i2
return 0 ;

## Lecture 7:

Machine generated alternative text: HOP Primitives
Higher order programming simplifies programming over
collections (lists, sets, bags, dictionaries)
Primitives of higher order programming
? ap : apply a function to every element of a collection
arid create a similar collection of results
? fold : combine all the elements of a collection via an
operator
? filter : remove from a collection the elements that do
not satisfy a predicate
? zip : create a collection of pairs, each pair being made
up of elements of the same rank in two input collections
They form a very useful abstraction barrier

## Lecture 8:

Machine generated alternative text: VAL Code Structure for Procedures
.
.
.
Caller
caller prologue
save caller registers
perform call
de-allocate arguments
restore caller registers
Callee
callee prologue
save callee registers
allocate local variables
body of procedure
callee epilogue
deallocate local vars
restore callee registers
return
Stack
caller registers
arguments
return address
callee registers
local variables
Record
bind actual arguments to formal arguments
set up return address
return_address: 4
caller epilogue
.
.
.
EBP
Activation
Slide 4 of 21
C52104 ? Lecture 8
October 6, 2011

Type: a collection of computational entities that share some common property

Strongly typed: all type consistency can be checked at compile-time; no need for run-time checks

Polymorphism: a symbol may have multiple types simultaneously

Forms of polymorphism:

* + Parametric polymorphism: function may be applied to any arguments whose types match a type expression involving type variables
  + Ad-hoc polymorphism (overloading): two or more implementations with different types are referred to by the same name
  + Subtype polymorphism: a subtype relation is defined between types; an expression with a given type can be used as argument anywhere where a subtype of the current type is expected