Evolution of programming languages

- Machine language
- Assembly language
- Sub-routines and loop (Fortran)
- Procedures and recursion (Algol, Pascal, C)
- Modules (Modula-2, Ada)
- Objects (Simula, Smalltalk, C++,Java)
- Declarative programming languages (Prolog, CLP, Lisp, ML, Haskall)

Why are there so many languages

- Evolution
  - Procedural → structural → object-oriented
- New paradigms and applications
  - Logic languages (Prolog, CLP) for complex data and knowledge processing
  - Functional languages (Lisp, ML, Haskell) for symbolic computation
  - Scripting languages (JavaScript, Pearl, Tcl, Python, Ruby, XSLT) for Web-based data processing
- Personal preferences

What makes a language successful

- ✓ Expressiveness
- ✓ Availability of implementations
- ✓ Efficiency
- ✓ Productivity
- ✓ Industrial sponsorships

Why study programming languages

- Understand language features and concepts at a higher level
- Improve the ability to choose appropriate languages
- Increase the ability to learn new languages
- Simulate useful features

Why study language implementation

- Understand how languages are specified and implemented
- Understand obscure phenomena
- Write better-style and efficient programs
- Design and implement domain-specific languages

Programming language spectrum

- Declarative
  - Logic and constraint-based (Prolog, CLP(FD))
  - Functional (Lisp/Scheme, ML, Haskell)
  - Dataflow (Id, Val)
  - Template-based (XSLT)
  - Database (SQL)
- Imperative
  - von Neumann (C, Ada, Fortran, Pascal,…)
  - Scripting (Perl, Python, PHP,…)
  - Object-oriented (Smalltalk, Effel, C++, Java, C#)

Imperative

- Features
  - Variables are mnemonics of memory locations
  - Assignment statements
  - goto
  - Iterative constructs

Stack in C

```
typedef struct NodeStruct {
  int val;
  struct NodeStruct *next;
} Node, *NodePtr, *List;

typedef struct StackStruct {
  int size;
  List elms;
} Stack, *StackPtr;
```

Stack in C (Cont.)

```
void stack_push(StackPtr s, int x){
  s->size++;
  lst_add(&s->elms,x);
}

int stack_pop(StackPtr s){
  if (s->size==0){
    error("empty stack");
  } else {
    s->size--;
    return lst_remove(&s->elms);
  }
}
```

Object-oriented

- Features
  - Abstract data types
  - Inheritance and overriding
  - Polymorphism
  - Dynamic binding

Stack in Java

```
import java.util.LinkedList;
class MyStack {
    private LinkedList<Integer> elms;

    public MyStack() {
        elms = new LinkedList<Integer>();
    }
    public void push(int x) {
        elms.addFirst(x);
    }
    public int pop() {
        if (elms.size()==0){
            throw new RuntimeException("Empty stack");
        } else return elms.removeFirst();
    }
}
```

Stack in C#

```
using System;
using System.Collections.Generic;

class MyStack {
    private LinkedList<int> elms;

    public MyStack() {
        elms = new LinkedList<int>();
    }
    public void push(int x) {
        elms.AddFirst(x);
    }
    public int pop() {
        if (elms.Count==0){
          throw new System.Exception("stack underflow");
        } else {
          int tmp = elms.First.Value;
          elms.RemoveFirst();
          return tmp;
        }
    }
```

Stack in C++

```
class Stack {
public:
  Stack();
  void push(int);
  int pop();
private:
  list<int> elms;
};

Stack::Stack(){
}
```

```
void Stack::push(int x){
  elms.push_front(x);
}

int Stack::pop(){
  assert(!elms.empty());
  int x =  elms.front();
  elms.pop_front();
  return x;
}
```

Functional

✓ Features

- Single assignment variables (no side effects)
- Recursion
- Rule-based and pattern matching (ML, Haskell)
- High-order functions
- Lazy evaluation (Haskell)
- Meta-programming (Scheme)

Stack in Scheme

```
(define stack_push
     (lambda (s x) (cons x s)))

(define stack_peek
     (lambda (s) (if (eq? s ())
                     (raise "empty stack")
                     (car s))))

(define stack_pop
     (lambda (s) (if (eq? s ())
                     (raise "empty stack")
                     (cdr s))))
```

Stack in Haskell

```
stack_push s x = (x:s)

stack_peek (x:_) = x

stack_pop (_:s) = s
```

Stack in SML/NJ

```
fun stack_push s x = (x::s)

fun stack_peek (x::s) = x

fun stack_pop (_::s) = s
```

F#

```
let stack_push s x = x :: s

let stack_peek s =
   match s with
   | x :: _  -> x

let stack_pop s =
   match s with
   | _ ::s1 -> s1
```

Logic & constraint-based

- Features
  - Logic variables
  - Recursion
  - Unification
  - Backtracking
  - Meta-programming

Stack in Prolog

```
stack_push(S,X,[X|S]).

stack_pop([X|S],X,S).
```

Stack in Picat

```
stack_push(S,X) = [X|S].

stack_peek([X|_]) = X.

stack_pop([_|S]) = S.
```

Implementation methods

- ✓ Compilation
  - Translate high-level program to machine code
    - Slow translation
    - Fast execution
- ✓ Pure interpretation
  - No translation
    - Slow execution
    - Becoming rare
- ✓ Hybrid implementation systems
  - Small translation cost
  - Medium execution speed

Review questions

- Why are there so many programming languages?
- What makes a programming language successful?
- Why is it important to study programming languages?
- Name two languages in each of the following paradigms: procedural, OOP, logic, and functional.
- What are the features of OOP languages?
- What are the features of functional languages?
- What are the features of logic languages?