ELTE - PPKE informatika tananyagfejlesztési projekt TÁMOP-4.1.2.A/1-11/1-2011-0052

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.

Advanced Programming Languages

Zoltán Porkoláb, PhD.
Associate Professor
Eötvös Loránd University, Faculty of Informatics
Dept. Of Programming Languages and Compilers

2013.

The book

- The book: Advanced Programming Languages
  - To be published in 2013
  - Editor: Judit Nyéky-Gaizler, PhD
  - Almost 20 authors, mostly from Eötvös LorándUniversity, Faculty of Informatics
  - 1080 pages, 17 chapters + Appendix
- The predecessor book:
  - 2003, in Hungarian

The purpose of this training

- The book
  - Generic concepts in programming languages
  - Unified terminology
  - Cross-reference between chapters
- The training
  - Summarize most important language features
  - Recap key concepts
  - Base of programming language class/training

Content (1)

- Language design
- Lexical elements
- Control structures
- Scope and Life
- Data types
- Composite types
- Subprograms
- Exception handling

Content (2)

- Abstract data types
- Object-oriented programming
- Type parameters (Generics)
- (Correctness in practice)
- Concurrency
- Program libraries
- Elements of functional programming languages
- Logic programming and Prolog
- (Aspect-oriented programming)

Language design

Language design

- Key concepts:
  - Syntax, Semantics, Pragmatics
  - Implementation
  - Programming language evolution
  - Language categories
  - Language design
  - Standardization

Syntax, Semantic, Pragmatics

- Syntax
  - The correct grammar of the language
- Semantic
  - The meaning of a syntactically correct phrase
- Pragmatics
  - How to use the given phrase for a useful purpose

Implementation

- Compilation
  - Phases: (Preprocessing), Compiling, Linking
  - Static or dynamic linking
  - Generates HW and OS-specific executable
  - Effective optimizations
- Interpretation
  - Faster developing process
  - Less correctness-checking possibilities

Implementation

- Hybrid model
  - Compiler generates platform independent intermediate code
  - Intermediate code executed by “virtual machine”
  - Fair correctness checking and optimization
  - More optimization: Just-in-time compilers
- Samples
  - Pascal P-code, Java virtual machine, MS IL

Evolution of the programming languages

- Early attempts
  - Computation of Bernoulli Numbers for the Analytical Engine – notes from Ada Lovelace
  - Plankalkül (Zuse, 1943-45) – relational algebra
  - Hard-wired machines (1940 -)
- Raising the abstraction level
  - Machine code (1945-50)
  - Assembly (1950-)

Evolution of the programming languages

- Early high level languages
  - FORTRAN (1956) – Math expressions
  - LISP (1957) – Functional
  - ALGOL (1958-60) – First block structure
  - COBOL (1960-) – Detailed data description
  - PL/1 – Union of all existing features
  - Basic – (Kemény), Simplification for education

Evolution of the programming languages

- New directions for better abstraction
  - Simula 67 (1967) – First OO language
  - Algol 68 (1968) – More precise specification
  - Pascal (1970) – Educational purposes
  - C (1971) – HW abstraction for system programming
  - Smalltalk (1971-) – Pure OO language
  - Prolog – First Logic programming language
  - ML – Statically typed functional language

Evolution of the programming languages

- Towards better modularization
  - Modula-2 (1978) – (Pascal) Better modularization
  - ADA (1977-) – Programming safety critical systems
  - C++ (1980) – C + Simula 67 + Algol 68
  - Oberon (1986) – Modula 2++
  - Objective C – Object based, dynamic

Evolution of the programming languages

- Towards faster development – Scripting languages
  - Perl, Python, Ruby, PHP (1985-)
- … and hybrid languages
  - Java - easy to use and deploy (virtual machine)
  - C#
  - Scala
- Just now: towards many-core & multicore systems
  - OpenCL, Go, ...

Programming language classification

- Imperative (procedural)
- Applicative (functional)
- Rule-based or logic
- Object-oriented
  - Object-based, class-based
- Concurrent
- Scripting (dynamic)

Language design concepts

- Well-defined syntax and semantics
- Expressivity
- Orthogonality
- Generality
- Modularity
- Portability
- Easy to learn
- Performance

Lexical elements

Lexical elements

- Key concepts:
  - Compilation units
  - Lexical elements, character sets
  - Delimeters, strict and non-strict format languages
  - Identifiers
  - Keywords, reserved words
  - Literals (number, character, string, …)
  - Comments

Control structures

Control structures

- Key concepts:
  - Sequences
  - Transfer of control
    - Conditional
    - Unconfitional
  - Subprogram (function, subroutine) call
  - Return from subprogram
  - (Exceptions)

Representing the control structure

- Sentence-like descriptions
- Flow diagrams
- D diagramss
- Block diagrams
- Structograms

Imperative (procedural) languages

Abstraction of von Neumann computer

- Variables representing the program state
  - Assignment = change state
- Execution is a sequence of state transitions
- Procedures: nesting state + state transition

Declarative and Functional languages

No method of execution is specified

- Specification of the problem to solve
  - SQL, Prolog
- In functional languages
  - The problem specification is to solve a (pure) function
  - Input/output is considered as „side effect”

Assignment

- Statement in earlier languages
- Expression in modern languages
  - Can be nested
  - Has return value
  - User may redefine semantics (C++ operator=)
- Some languages (CLU) has multiply assignment
- Implicit conversions are involved

Unconditional transfer of control

- Go To statement
- …is considered harmful (Edsger W. Dijkstra, 1968)
- Sometimes still used in modern languages
  - Break and continue in C
  - Return from the middle of a subprogram
- Call of subprogram and return
  - Recursion

Conditional transfer of control

- Arithmetic GOTO in Fortran
- Branching structures
  - If, elif, else
    - Dangling if
  - Switch/case
    - Default case

Loops

- Loops on condition expression
  - While, do-while, for(expr1;expr2;expr3) in C
- Iteration over an integer range
- Iteration over a value range
  - Foreach in C#, for (expr) in C++
- Iteratiors
  - Abstraction over control structure

Scope and Life

Scope and Life

- Key concepts:
  - Scope
    - Identifyer, declaration, definition
    - Block structure, visibility
  - Life (or lifetime)
    - Construction, destruction
    - Garbage collector, Memory leak
    - RAII, Smart pointers

Scope and Life

- Scope
  - Static feature – compilation time
  - The mapping between names (identifiers) and program objects (types, functions, variables)
- Life (or lifetime, life span)
  - Dynamic feature – during runtime
  - The time between the creation and destruction of the object
- Related, but not identical features

Scope types

- The entire program
    - perhaps more compilation units
- One compilation unit
- A type or class
- A namespace
- A subprogram
- A block of code

Scope rules

- Scopes may overlap
  - Internal scope hides external one
  - In some languages: syntax error
- Important difference between
  - Hiding (of a name in external scope)
  - Overriding (of virtual function)
  - Overloading (between functions with same name but different signature)

Life, lifetime, life span

- Static
  - Memory allocated at the beginning of program
  - Memory deallocated at the end of the program
- Automatic
  - Memory allocated when control enters the block
  - Memory deallocated when leaving the block
- Dynamic
  - Allocation controlled by the programmer
  - Deallocating manually or by garbage controller

Static life

- Memory allocated at the beginning of the program
- Life keeps until the end of the program
- Nasty details:
  - Java: construction when loading the class
  - C++: no creation order between compilation units
  - C++: Static initialization issue, when constructors of static variables refers to each other. Use Singleton pattern!

Automatic life

- To reuse memory between disjunct subprograms
- Objects are allocated in the stack
- Mostly used for local variables and temporaries
- Objects constructed at entering the code block where the object is declared (in declaration order)
- Objects are destructed when leaving the block
- Nasty details:
  - Sometimes we have reference to variable after automatic life finished (e.g. return pointer to it)

Dynamic life

- Construction and destruction (mostly) controlled by the programmer
- Objects are allocated in the heap/free memory on programmer request
- In some languages deallocation is on request
- In other languages: garbage collection
  - Difference between destruction and finalization
- Heap operations are very slow in-memory activities

Dynamic life

- Memory leak: when allocations and deallocations do not match
  - Can happen even with garbage collection
  - Usually happen when no garbage collection
  - Throwing exceptions is a typical source of issue
- RAII – Resource allocation is initialization
- C++ smart pointers using RAII
  - Ownership or reference counting strategy

Data types

Data types

- Key concepts:
  - What is a data type
  - Specification and realization
    - Invariants
  - Type system
    - Strongly typed, graduate typed, typeless
    - Type inference
  - Type conversions

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
ÚJ SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.

Data type categories

- Primitive/built-in types
  - Scalar types
    - Integral types
    - Floating point types
    - Characters
    - Enumerations
  - Pointers
    - Pointers to objects, pointers to subprograms

Composite types

Composite types

- Key concepts:
  - Abstract constructions
    - Cartesian product types
    - Union types
    - Iterated types
  - Type equivalence
    - Name equivalence
    - Structure equivalence
    - Declaration equivalence

Cartesian product

- Type-value set: T1 x T2 x T3 x … x Tn
  - Widely supported in languages: record, struct, ...
- Operations
  - Type/Field selection
  - Assignment
  - Equality check
- Language specific
  - Variadic record
  - Default values

Union

- Type-value set: T1 ᴗ T2 ᴗ T3 ᴗ … ᴗ Tn
  - Less support in languages: union, variant
  - Some OO languages use inheritance instead
  - Tagged or free union
- Operations
  - Type/Field selection
  - Assignment
  - Type selection (in some languages)

Iterated types: Array

- Type-value set: T x T x T x … x T
  - Full support in languages: array, ...
  - Length may variadic (given at runtime) or static
  - In some languages arrays „know” their lenght
- Operations
  - Selection based on index value
  - Assignment is not fully supported
  - In C special relation between pointers and arrays

Iterated types: Set

- Type-value set: 2^T
  - Partial support in (mostly Pascal-like) languages
  - Otherwise implemented as library type
- Operations
  - Assignment
  - Equality check
  - Set operations (push, pop, has, ...)

Iterated types: Set

- Type-value set: 2^T
  - Partial support in (mostly Pascal-like) languages
  - Otherwise implemented as library type
- Operations
  - Assignment
  - Equality check
  - Set operations (push, pop, has, ...)

Other iterated types

- Hashtables
  - Key-value pairs
  - Mostly in script languages (Perl)
  - Otherwise implemented as library type (C++, Java)
- Multisets/bags
  - Key-counter pairs
  - Usually implemented as library type

Subprograms

Subprograms

- Key concepts:
  - Formal and actual parameters
  - Parameter passing methods
  - In, out, and in-out parameters
  - Overloading
  - Subprograms as parameters
  - Corutines

Subprograms

- Reusing existing code parts (since Babbage!)
- Positive effect on code quality
  - Reusablility
  - Readability
  - Changeability
  - Maintainability
- Procedures and functions

Subprogram structure

- Function signature
  - Name
  - Parameter list
    - Formal parameters: at subprogram definition
    - Actual parameters: at subprogram call
  - Const, volatile modifiers are part of the signature
- Return value (for functions)
- Usually a single entry point (exceptions, like PL/I)
- Potentially multiply return points

Calling subprograms

- Explicit call statement with keyword, like CALL f()
- Just write a call expression, like x = f()
- Actual parameters match with formal parameters
  - Either prameters matching by name
  - Or parameters are passed using formal name
- Default parameters (if any) are evaluated at calling site
- In some languages () can be omitted at calls with no actual parameter

Parameter list

- Sometimes we have variadic parameter list
  - printf(const char *fmt …)
- Sometimes we have default parameters
  - void f(int x = 1)
- Sometimes we overload functions on parameters
  - void f(double x) and void f(int x)
- Sometimes we overload on modifiers:
  - void F(int* x) and void F(const int* x)
- In OO languages we pass hidden parameter “this”

Parameter passing by value

- Actual parameters are copied into the subprogram
- Formal parameters acting like local variables
- Best separation of caller and callee
  - Formal parameter identifies different memory area than the actual parameter, changing them has no effect on caller
  - Parameters transfer information only into callee
  - Only return value transfer information to caller
- Out parameters can be simulated by passing pointers

Parameter passing by reference (address)

- Actual parameter addresses are used in subprogram
- Formal parameters acting like global variables
- Weak separation of caller and callee
  - Formal parameter identifies the same memory area than the actual parameter, changing them has permanent effect on caller
  - Parameters may transfer information in and out
- Issues when actual parameter is an expression not identifying a memory area, like: CALL F(k+1)

Parameter passing by result

- Modification of pass by value for implementing output parameters
- Actual parameters are copied into the subprogram
- Formal parameters acting like local variables
- When the subprogram returns, value of formal parameters are copyed back to actual parameters
- Good separation of caller and callee
  - Modification of formal parameters has no effect to caller until subprogram returns

Parameter passing by name

- Actual parameters not identifying any memory region (expressions, like 3+4) are passed by value
- Actual parameters referring to memory region (expressions, like 3+t[i]) are passed by address
  - Every time a formal parameter is referred during subprogram execution, the expression specified as actual parameter is re-evaluated.
- Weak separation of caller and callee
  - Modification of actual parameters has immediate effect to the subprogram execution

Textual substitution of parameters

- Used mainly in simple scripting languages and macros
- Weak separation of caller and callee
  - No separation of caller and callee
  - Dangerous side-effects may happen, like
    - Multiply evaluation
    - Precedence hijacking

Overloading

- The same name can denote multiply subprograms
- Compiler selects the appropriate subprogram
  - Overload resolution happens in compile-time
  - Compiler flags compile error if unable to select
- Overloading happens on
    - Number of parameters
    - Types of parameters
    - Modifyers (like const and volatile)

Operator overloading

- Operators – when programmer can define them – are acting like functions with special names
- ADA, C++, C# allows operator overloading, Java not
  - Arity and precedence of operators are usually fixed

Passing subprograms as parameters

- Subprograms are first calss citizens in functional programming languages
  - They can be passed as parameters
- In imperative and OO languages this is not general
  - Funtion pointers in C/C++
  - Function objects and lambda functions in C++
  - Modula-3: closure carrying the environment where the subprogram will be executed

Corutines

- Corutines are subprograms executed in a symmetric control model instead of caller-callee distinction
  - The corutin can pass back the control without finishing its task – dispatch
  - Local variables are kept alive
  - Corutin can resume its execution from the place where it was last dispatced
- They mimic parallelism – sometimes decreaseing program complexity

Exception handling

Exception handling

- Key concepts:
  - Handling runtime errors
  - Separation of error handling concepts
  - Exception safety
  - Checked and unchecked exceptions
  - Grouping exceptions
  - Polymorphic exceptions

Basic concepts

- Runtime errors break normal execution flow
  - Error handling code crosscuts normal execution
  - Separation of error handling is welcome
- Exception may raised (throw, signal) by SW or HW
- Exceptions are catched by specific handlers
- Exceptions can carry information from the site of error to the place where we can handle it
  - In modern languages exceptions are objects of arbitrary type – we can define our type too

Exception safety

- As exceptions break the normal flow of execution
  - Object can be left in undefined state
  - Resources may not be deallocated
  - This can cause inconsistency, like memory leak
- In Java and C# finally blocks are executed before the control leaves a block
- In C++ RAII is used to avoid memory leak
  - Nothrow, strong, and basic garantees

Checked exceptions

- An exception throable from a subprogram or class is part of the interface of that module
- Checked exceptions in Java
  - Static (compile-time) checking whether the exception has been properly handled
  - There are non-checked run time exceptions too
- In C++ the compiler has less possibility for checks
  - In C++11 there is a __nothrow attribute and operator which is evaluated in compile time

Grouping exceptions

- In modern OO languages exceptions are common objects from arbitrary types
- Custom types can be defined for custom exceptions
- Inheritance hierarchy can be used to group exceptions
    - The handler of base class catch derived exceptions
    - Always throw the most derived type of exception
    - Exception objects can be polymorphic

Abstract Data Types

Abstract Data Types

- Key concepts:
  - Modular design
  - Language support for modularity
  - Representation hiding
  - Separation of specification and implementation
  - Generalized program schemes

Modular design

- In modern programming languages modular design is a key concept.
- Criteria of modular design:
  - Modular decomposition
  - Modular composition
  - Modular intelligibility
  - Modular continuity
  - Modular protection

Modularity

- Language support of modules
- Few interconnections
- Weak interconnections
- Explicit interfaces
- Information hiding
- Open and closed modules
  - Open: extendable (like C++ namespaces)
  - Closed: reachable via interface, used unchanged

Modularity

- Reusability
- Variety of types
- Variety of data structures and algorithms
- One type – one module
- Representation independence

Language Support for Modularity

- Procedural languages
  - C: #include
  - Modula-3: modules, export, import, generic
  - Ada: package – open modules support
- Functional languages
  - Signature description in ML, can be reused
    - By embedding
    - By specialization

Object-oriented languages

- Address open/closed modules with inheritance
- Package (Java as example)
  - Package visibility
  - Import, imported names should be fully qualified
    - Single-type import
    - On-demand import

Representation hiding

- Opaque type
  - Handlers in C, Pascal
  - ADA: private type
  - CLU: abstract data types
- Visibility levels
  - Private
  - Protected
  - Public

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
ÚJ SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.

Specification and Implementation

- Enables the modules to be developed separately
- Supports modification in implementation
- Implementation can be delivered in binary
- Supports separation:
  - C and C++ header files / source files separation
  - Mapping to pointers / C++ PIMPL strategy
- Not supports physical separation
  - Eiffel, Java

Generalized program schemes

- Data parameters
- Type parameters: Generic, Template
  - Type erasure
  - Instantiation
- Subprograms as parameters
  - Function pointers, functors, lambdas
- Higher level structures as parameters
  - Scala

Object-oriented programming

Object-oriented programming

- Key concepts:
  - Class and Object
  - Constructing and destructing objects
  - Encapsulation
  - Data hiding, interfaces
  - Class (static) data and method
  - Inheritance
  - Polymorphism

Class and Object

- Object
  - Independent units of the reality we model
  - Inner state (represented by attributes)
  - Response for the messages received (behavior)
  - Each object has its identity
- Class
  - Group of objects with similar attributes and behavior
  - Each object is an instances of a class

Construction, destruction

- Life cycle of objects: born, live, die
- Construction
  - Set the initial state to fulfill type invariant
  - Constructor: usually a public method
- Destruction
  - Sometimes need to clean-up resources
  - Destructor (C++)
  - Finalize method (called by Garbage Collector)

Encapsulation

- Considering data structure and the operations on it as a single unit and hide them from outer word
- Specification gives the outer description
  - Value-set of the type
  - Behavior: methods
- Implementation
  - Data representation
  - Method implementation

Data hiding, interfaces

- Encapsulation means objects hides implementation
- To communicate with the outside word: interface
  - Interface: previously defined set of messages
  - Object can be touched only using the interface
  - Interface should be minimal
- Visibility
  - Public, protected, private
  - (C++) friends, (Eiffel) selective access

Class (static) data and methods

- Normally we work with objects and their methods
  - For such instance methods we pass Self/This
- Sometime we use data and methods not connected to any objects – called class (static) data and method
  - Cannot access instance members
  - Cannot call instance methods (not receiving Self)
- C++, C#, Java: static members, static methods
- Scala: object – a singleton

Inheritance

- Generalization and specialization
- Inheritance
  - Super-class (base) and sub-class (derived)
  - Derived inherits base attributes and methods
  - New attributes and methods can be added
  - Cannot access instance members
  - Cannot call instance methods (not receiving Self)
- Inheritance can be single and multiply
  - Some lang multiply inheritance only for interfaces

Inheritance types

- Derived class access public and protected of base
- Inheritance normally extend base interface
  - Java keyword: extend
- In C++ there are three kind of inheritances
  - Public: extend interface as Java
  - Protected: convert base interface to protected
  - Private (default): Hide the interface of base

Polymorfism

- Extending base interface: substitutability
- Liskov Substitution Principle
  - Subclass objects can appear in place of super
- Static and dynamic (run-time) type of object
  - Polymorphism: in OO = subtype polymorphism
- Methods can be redefined in subclass
  - Static redefinition (based on static type)
  - Dynamic redefinition (based on run-time type):
    - Virtual functions, dynamic dispatch

Abstract class, interface

- Representing common abstractions
- No objects are instantiated from abstract class
  - Common data for all subclasses
  - Protocol: for redefine in subclasses
- C++ abstract class
  - Pure virtual
- Java
  - Interface: no data
  - Abstract class

Multiple inheritance

- Mostly for unifying multiply interfaces
- In many languages only for interfaces
  - Java, C#
- In C++ works by default
  - Resolving name conflict with scope operator
- Diamond-shape inheritance
  - Common base class inherited multiply times
  - Scala traits

Type parameters

Type parameters

- Key concepts:
  - Control and data abstraction
  - Taxonomy of Polymorphism
  - Generic contract model
  - Instantiation
  - Type erasure

Abstraction

- Control Abstraction
  - Procedural (C): function pointers
  - Object-oriented (Java): classes with “doit” func.
  - Generic (C++): functors – function objects
- Data abstraction
  - Type parameters
  - Opaque types

Taxonomy of Polymorphism

- Universal
  - Parametric
  - Inclusion
- Ad-hoc
  - Overloading
  - Coercion

Generic Contract Model

- Constraining type parameters for data or method
- Constrained generics
  - Java – supertype, subtype relationships
  - Ada – with clause
- Unconstrained generics
  - C++ templates – no restriction on type parameter
  - C++14 Concepts (light)
- Type class in functional languages

Instantiation

- Actual type parameter substitues formal parameter
- New code generated – instantiation
  - On demand (C++)
  - Manually (ADA)
- User defined specialization in C++
- Template metaprogramming – executing algorithm in compile-time

Type erasure

- Actual type parameter substitued by common heir
- Most famously used in Java
- Only one code serves all type parameters
- Compiler ensures type safety on back-conversion
- Auto-boxing unboxing for built-in types
- No specialization

Concurrency

Concurrency

- Key concepts:
  - ”The free lunch is over”
  - Why to write concurrent applications?
  - Amdahl's law
  - Syncronization
  - Process vs. Thread
  - Sample: MPI and Java

The free lunch is over

- Moore's law
  - Number of transistors
  - Processing speed

    doubles in every 24 month
- Free lunch: the same program will run faster by time
- Trend on speed has broken:
  - The free lunch is over (Herb Sutter, 2004)
  - Make programs faster should utilize concurrency

Area of concurrent programming

- Responsive user interfaces
- Multi-user server solutions incl. databases
- Web servers, and web browsers
- Multicore systems to utilize all resources
- Scalable applications
- More intuitive programming schema

Popular fallacies on Concurrency

- If it is concurrent it is quicker.
- Program structure does not matter.
- Easier to write a sequential prototype and then rewrite it as a parallel version.
- I do not need to care concurrency.
- Concurrency is easy. At least easy to debug if I make mistakes.

Amdahl's law

The performance gain by paralellizing an application is heavily bounded by the ration of concurrently executable parts that are independent of each other

Data and Instruction

- Single instruction
  - Single Data Stream SISD
  - Multiply Data Stream SIMD
- Multiply Instruction
  - Single Data Stream MISD
  - Multiply Data Stream MIMD

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
ÚJ SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.

Syncronization

- Deadlocks
  - happen if all these conditions state (Coffman conditions):
    - Mutual exclusion
    - Hold-and-wait locking
    - No preemption
    - Circular dependencies
- Starvasion

Syncronization techniques

- Critical section
  - Set of instructions where the execution is restricted to a single thread/process.
  - Entry protocol, exit protocol
- Busy waiting
- Semaphore
- Monitor
- Conditional critical section

Concurrent execution units

- Process vs thread
- Process
  - Own address space
  - Expensive to start and switch
- Thread
  - Own execution thread, shared address space
  - Thread local memory - own stack
  - User / system thread

Sample: MPI

- Message Passing Interface – language independent
- Inter-thread point-to-point communication
- Handle tasks in groups
  - Creating tasks
  - Communication methods
  - Communication in group

Sample: Java

- Runnable interface and Thread class
- Thread groups
- Concurrent API
- Concurrent collections
- Executor framework
- Thread pools
- Syncronized

Program libraries

Program libraries

- Key concepts:
  - Requirements against libraries
  - Procedural programming
  - Object-oriented programming
  - Generic programming

Library requirements

- Correctness
- Efficiency
- Reliability
- Extensibility and maintainability
- Reusability
- Portability

Procedural library design

- Set of type definitions and functions
- Separation of interface and implementation
- Most frequently tasks are covered:
  - Standard I/O: file handling
  - Memory manipulation
  - String library
  - Mathematical functions

Object-oriented library design

- Class hierarchy
- Optimal service size < 80
- Abstract types with interface
  - Fat vs. narrow interfaces
  - Handle classes, proxy classes
- Expression problem:
  - Easy to extend with new data (class)
  - Hard to extend with new service
  - Visitor pattern

Generic programming library design

- First introduced in C++ as STL (Stepanov&Musser)
- Orthogonal separation of data and algorithm
- Containers (parametrized by types)
- Algorithms without knowing data representation
- Iterators connecting containers to algorithms
    - Hierarchy of iterators
- Functors
- Adaptors

Functional Programming

Functional Programming

- Key concepts:
  - Mathematical foundation: Lambda calculus
  - Structure: function definitions, starts expression
  - Referential transparency
  - Pattern matching

Mathematical foundation

- Lambda calculus (Church 1932-33)
  - Equivalent to Turing Machine
- Evaluation of (mathematical) functions
- First implemented: LISP (1957)

Functional program structure

- Type, type class and function declarations
- Program execution = evaluate start expression
- Rewriting system: rewriting start expression determined with the computation model
- Pure (side-effect free) functions
- Referential transparency (no assignment)
- Strong static typing
- Pattern matching

Functional program structure

- Recursive function application
- Currying, partial function application
- Higher order functions
- Evaluation strategy
  - Lazy evaluation
  - Eager, strict evaluation
- List comprehension

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
ÚJ SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.

Logic Programming

Logic Programming

- Key concepts:
  - Algorithm = Logic + Control
  - Logic programs
  - Prolog

Logic programming

- Roots: Automated theorem prover (suggested by Hilbert) has two components:
  - Logical (declarative) description of problem
  - Control component of deduction or computation
- Axiom:
  - Fact
  - Rule (incl. recursive rules)
- Search trees

Prolog

- Robert Kowalski (~1970): Theoretical foundations
- Alain Colmerauer (1972): PROLOG
  - Implemented first as an interpreter
- David Warren (~1976): First effective compiler
- Data structure: terms
  - Set of predicates and a query or goal
- The prolog machine

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638
MAGYARORSZÁG MEGÚJUL
SZÉCHENYI TERV
A projekt az Európai Unió támogatásával valósul meg.