1005 1663
638

Subject : Introduction

Date : 26 / 09 / 2020

- What's a programming language? "Conceptual Universe" → Kavramsal Evren

It is an artifical language designed to communicate instructions to a machine, particularly computer

It is providing framework for problem solving and useful concepts.

CPU can run with Assembly Code

* Understand languages we use, by comparison
* Appreciate for new pr. methods, tools
* Critical thought (Identify properties of language)

Trends:

→ Commercial trend → Java, C#

→ Teaching trend → Java replaced C, Objective C

→ Modularity → Java, C++

Look D Language ?

2008 → Decrease

Increasing use of type safe languages

Scripting languages, other languages for web app.

Objective C: IOS, Android etc.

D - Lang.

- H. level Constructs for great modeling power
- High Performance
- Static Typing
- Direct interface to the OS API's and hw
- Blazingly fast compile-times
- Memory-safe subset
- Maintainable, easy to understand code
- Gradual Learning Curve (C-like syntax, similar to Java and etc.)
- Compatible with C app binary interface
- Limited compatibility with C++ app
- Multi paradigm (Imperative, OO, etc)
- Built in error detection (contracts, unit tests)

Subject: Paradigms Date: 26 09 2020

* Evaluating PL
* Programming Paradigms
* Compilation Process

Language Evaluation Criteria

- Readability
- Writability
- Simplicity, orthogonality → sadeleşme olması
- High expressive power, flexibility
- Reliability
- Safety
- Cost (Creation, execution, maintenance)

- Readability
  - Overall simplicity
    - Too many features ✗
    - Multiplicity of features ✗
  - Control Statements
  - Data Type and Structures
  - Syntax Considerations
  - Orthogonality
    - Makes the language easy to learn and read
    - Meaning is context independent

- Writability
  - Simplicity and orthogonality
  - Support for abstraction
  - Expressivity
  - Development environments

- Simplicity
  - It improves read/writability
  - Large language takes more time
  - Feature multiplicity is often confusing
  - Operator overloading can lead confusion
  - Some languages, can be "too simple"

Subject: Paradigms (Continue) Date: 26./09./2020

- Orthogonality
  - A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations
  - Fewer exceptional rules = conceptual simplicity (Kavramsal basitlik)
  - Tradeoffs with efficiency
  - It improves read/writability
- Support for Abstraction
  - Data → Programmer-defined types and classes
    Class libraries
  - Procedural → Programmer-defined functions
    Standard function libraries
- Reliability
  - Program behavior is the same on different platforms ?
  - Type errors are detected (C vs. ML)
  - Semantic errors are properly trapped (C vs C++)
  - Memory leaks are prevented (C vs. Java)
  - Type Checking, Exception Handling, Aliasing (Takma ad), Readability and Writability

*p++ = *q++ ?

a=7 *p=a

b=8 *q=b

a=8 p=...adresi

b=8 q=8

Subject : Paradigms

Date : 26./09./2020

Efficient Implementation

- Embedded Systems (Real-time responsiveness)
- Web Applications (Responsiveness to users)
- Corporate (Kurumsal) Database Applications (Efficient search-upload)
- AI applications

Cost

- Programmer Training
- Software Creation
- Compilation
- Execution
- Compiler cost
- Poor reliability
- Maintenance (Onarım)

Others

- Portability
- Generality
- Well-definedness

What Is Computation

→ Solving a problem → function

→ A decision problem → well defined question about well-specified data that has a yes-no answer (For example: Primality / Asallık)

Partial and Total Functions

Programs define partial functions for two reasons

1- Partial operations

2- Nontermination

Subject: Paradigms (Partial and Total F.) Date: 26./09./2020

Computability

- Partiality: Recursive functions may be partial. They are not always total. A function may be partial because a basic operation is not defined on some argument or because a computation doesn't terminate
- Computability: PL can be used to define computable functions (We cannot write functions that are not computable in principle)
- Turing Completeness: All standard general-purpose pl give us the same class of computable functions
- Undecidability: Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable

Paradigms

A scientific paradigm is a framework

- Imperative (Zorunlu)
  - Procedural
  - Object-Oriented
- Declarative
  - Functional / Applicative
  - Logic
  - Mathematical

- In reality, very few languages are "pure"
- Programming paradigms, are the result of people's ideas about how programs should be constructed

Concurrent and Scripting
(Eş Zamanlı) (Komut Dosyası Oluşturma)

- Concurrent programming cuts across imperative, OO and functional paradigms
- Scripting is a very high level of programming
  - Rapid development
  - Often dynamically typed
  - Weakly typed (x can be assigned any time)

Subject: Paradigms Date: 29.09.2020

- Unifying Concepts (Birleştirici Kavramlar)
  - Types (Built-in and user defined) (kullanıcı)
    - Specifies constraints on functions and data
    - Static vs. dynamic typing
  - Expressions (arith., bool, string)
  - Function, procedures
  - Commands

- Abstraction and Modularization (soyutlama)
  - Re-use, extension of code are critically important in Soft. Eng.
  - Big Idea: Detect errors at compiling, not when program is executed
  - Type definition and declaration
    - Functions and data
  - ADT
    - Access local data only via a well-defined interface
  - Lexical Scope (Sözcük kapsamı)

- Static vs. Dynamic Typing
  - Static → Common in compiled languages. Each variable determined at compile time. Constains the set of values it can hold at run time
  - Dynamic → Common in interpreted (yorumlanmış) lang. Types are associated with a variable at run time. Type errors not detected until a piece of code is exec.

- Language Translation
  - Native Code Compiler → Produces Machine Code (Fortran, C, C++, SML)
  - Interpreter → Translates into internal form and immediately execute (read-eval-print) (Scheme-Haskell-Python)
  - Byte-Code → Produce portable bytecode which is exec on VM (Java)
  - Hybrid Approach → Source-to-source (C++ to C)

Java comp convert bytecode into native machine code when first exec

Subject : Compilation Date 29/09/2020

-- Compilation Process

- Compilation : Source code → relocatable object code (binaries)
- Linking : Many binaries → one " binaries
- Loading Relocatable → Absolute binary (with all code and data references bound to the addresses occupied in memory)
- Execution : Control is transferred to the first instruction of the program

-Phases of Compilation

- Preprocessing
- Lexical Analysis : Convert keywords, identifiers into a sequence of tokens
- Syntatic Analysis : Check token sequence is correct
- Generate abstract syntax trees (AST), check types
- Intermediate Code Generation
- Final code generation : Produce machine code.

Bytecode Compilation

- Compilation with interpretation (Idea: remove inefficiencies of read-eval-print)
- Similar to real machine opcodes, but they represents compiled instructions to a vm instead of real machine
  - Source code statically compiled into a set of bytecodes
  - Bytecode interpreter implements the VM
  - What way are bytecodes "better" than real opcodes?
    - If speed is the only important factor, go native; if the others (portability, security, size) go with bytecode

Language Interpretation

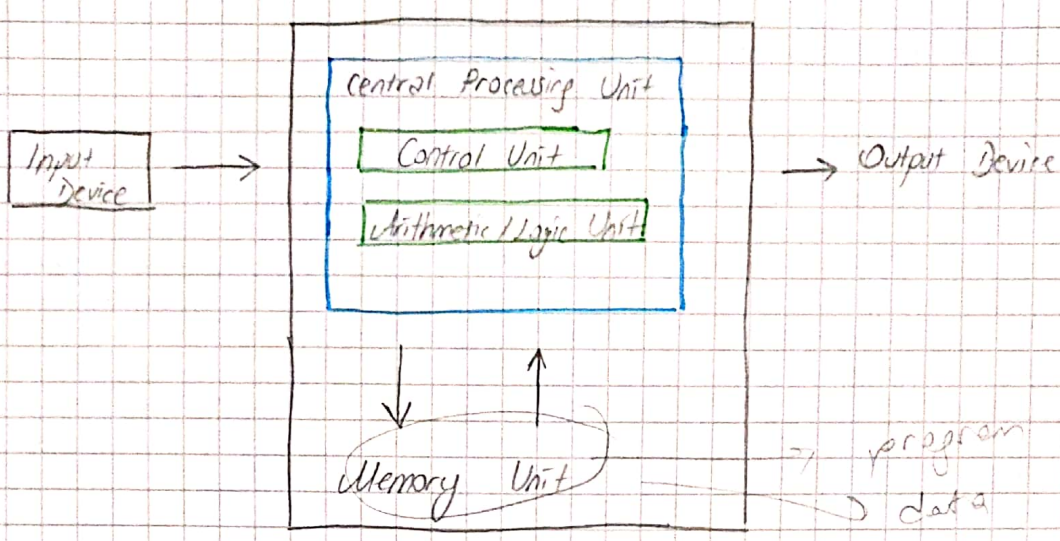
Read-eval-print

Opcode is a number of that represent single instr.

Bytecode : Representation of a prog

Subject: Von Neumann

Date: 29/09/2020



- PLs are high-level abstract isomorphic copies of von Neumann
- The isomorphism between VN programming languages and architectures

Program variables ↔ computer storage cells

Control statements ↔ computer test and jump instructions

Assignment " ↔ Fetching, storing instructions

Expressions ↔ Memory reference and arithmetic instructions

Subject: Syntax and Semantics

Date: 26/10/2020

Syntax: The symbols used to write a program
Semantics: The actions that occurs when prog. is executed

Input → Source Interpreter → Output

IP vs. Compiler

Source Compiler
Input → Target → Output

Typical Compiler

Lexical Analy. → Syntax → Semantic → Intermediate Code Generator → Code Optimizer → Code Generator

BNF : Integer → Integer | Digit ...

Parser verifies that this token stream is syntatically correct.

CFG For Floating Point Numbers

- <real number> = <integer part> '.' <fraction part> (1)

Non-terminal Non-terminal

4 non-terminal symbol!

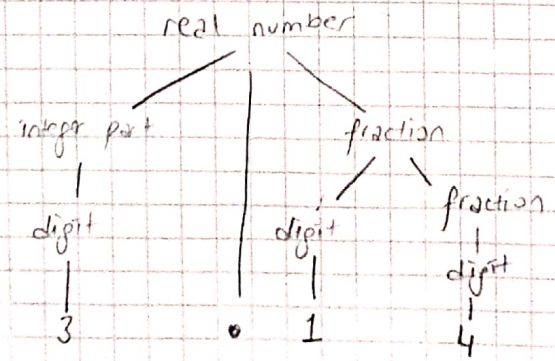
- <integer part> ::= <digit> (2) | <integer-part> <digit> (3)

- <fraction> ::= <digit> (4) | <digit> <fraction> (5)

<digit> ::= '0' (6) | '1' (7) | ... | '9' (15) → terminal symbols

↳ 15 different production rules

<...> ⇒ non terminals



Subject : Date : ...../...../.....

Grammars and Derivations

Application of a sequence of rules

<program> ::= begin <stmt_list> end

<stmt_list> ::= <stmt> ; <stmt_list> | <stmt>

<stmt> ::= <var> = <expr> (assignment)

<var> ::= A | B | C

<expr> ::= <var> + <var> | <var> - <var> | <var>

All the tree part should be terminal symbols

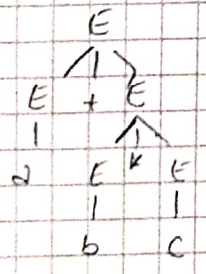
<assign> ::= <id> = <expr>

<id> ::= A | B | C

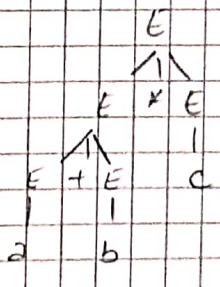
<expr> ::= <id> + <expr> | <id> * <expr> | (<expr>) | <id>

<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c

How to parse a + b * c



or



Subject : Variables

Date : 2.../11.../2020

- Variables
- Init°
- Constants
- Binding
- Type checking
- Scope

- Imperative languages are abstractions of von Neumann arch.

  * Memory
  * Processor

Variable: An abstraction of a computer memory cell or collection of cells

Characterized by 6 attributes

1- Name
2- Adress
3- Value
4- Type (Range of values, interpretation, operations)
5- Lifetime
6- Scope

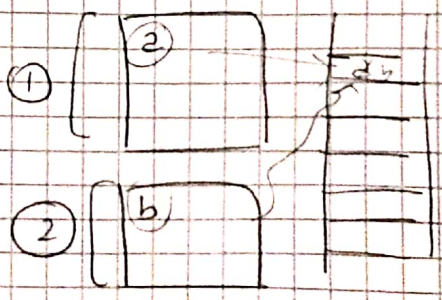
Case sensitivity

* Many lang are not case sens
* C, C++, Java are case case sens
* Clisp case sens

Variable Addresses may change in execution time

Variable Aliases (Nickname).

Instead, use dynamic allowtion



Subject :

Date : ../../..

Variable Types and Values

- Type - determines the
  - range of values
  - set of operations that are defined for var
  - interpretation of bit patterns
  - for floating point, determines the precision (possible)

The Concept of Binding

- The l-value = address
- The r-value = value
- Binding = association (between)
- Binding time : when l-value is associated with r-value
- Initialization :

- Possible Binding Times

* Language design time : bind operator symbols to operations
* " Imp " : " floating point type to a var
* Compile time bind a variable to a type in C or Java
* Load time : bind a FORTRAN 77 variable to a memory cell or bind static variable in C or Java
* Run time : bind a non static local variable to a memory

explicit / implicit declaration

Static vs. Dynamic Binding

↓	↓
First occurs before run time remains unchanged	First occurs during execution

Inferencing : coborrow

Type Bindings

- How is a variable's type specified?

Subject :

Date :...../...../.....

My Works (21-10-2020 Video)

Syntax and Semantics

- Syntax
  - Symbols used to write a prog
- Semantics
  - The actions that occur when a program is executed

Syntax

Lexical Syntax

Basic symbols (names, values, operators)

Concrete Syntax

Rules for writing expressions, statements, programs

Abstract

Internal representation of expressions and statements, capturing their meaning (i.e semantics)

Grammar Meta Language It consist of

1- Finite set of terminal symbols
2- " " " non-terminal "
3- " " " production rules
4- start symbol

} Backus-Naur Form (BNF)

- Language (possibly infinite)

Subject : Date :...../...../.....

Grammar for Unsigned Decimal Numbers

- Terminal sym : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Non-T : Digit, Integer

Rules

- Integer → Digit | Integer Digit
- Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Start sym

12 rules

Chomsky Hierarchy

- Regular Grammars
- Context-Free Grammars
- Context-Sensitive Grammars

1- Regular

Generate regular languages

Left - right regular ş.

Left

Non A → w A → Bw A → ϵ

Non terminal into terminal

Right

A → w A → wB A → ϵ

wB: w = terminal, B = non terminal

Subject :

Date :...../...../.....

Lexical Analysis splits source code into tokens

- Token = sequence of characters represent a single lexm sym

Identifiers : myVariable

Literals : 123 5.67 true

Keyword : char sizeof

Op. : + - * /

Punctuat. : ; , { }

Automatic Scanner Generation

Lexer or scanner recognizes and seperates lexical tokens

- Parser usually calls lexer when it's ready to process the next symbol

Deterministic Finate Automation (DFA)

- Graphs

Only one output

2- Context Free (CFG)

Describe concrete syntax

Parse Tree : Graphical Representation of Derivation

Tools : yacc, Bison

3 ay | adam / ay
5 kiz

Subject :

Date : ...../...../.....

Real number : : = <integer> . <fraction> (1) (<integer>: non-terminal)

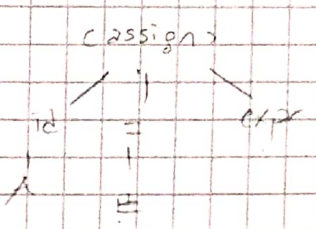
<integer> : : = <digit> (2) | <integer> <digit> (3)

<fraction> : : = <digit> (4) | <digit> <fraction> (5)

<digit> : : = '0' | '1' | '2' | '3' | . . . (15 rules)

A = B * (A+C)

A = <expr>



Parsers

Determine the input is syntactically correct

Produce parse tree

Classify parsers

- Top-down
- Button-up

LL Algo

Recursive descent - coded directly from BNF

Parsing table - do not implement BNF ↓

Both are version of LL

- Works on same subset of all CFG
- 1st L: left to right scan of input
- 2nd L: leftmost derivation is generated

Subject :

Date :...../...../.....

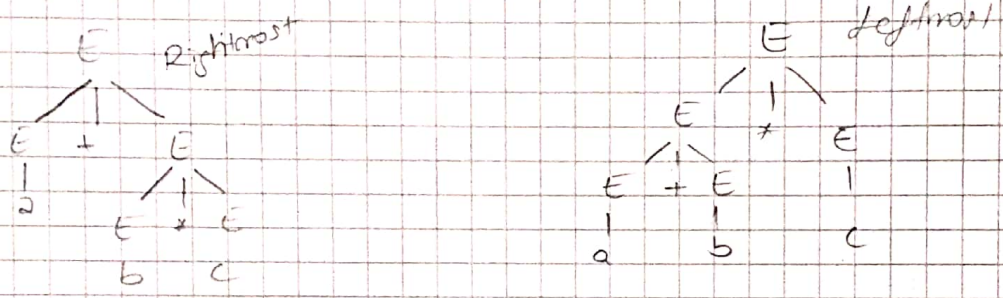
LR Parsing

- Bottom-up

L: Left to right
R: Rightmost der... in reversed

Syntatic Ambiguity

a+b*c



LR parsers most powerful than LL

Shift-Reduce Parsing

4 possible action

1- Shift → Move to the next input symbol to top of stack

2- Reduce → Replace the handle on the top of stack by non-termi...

3- Accept → Successful completion of parsing

4- Error

Subject :

Date : 30./11./2020

~~Scope~~ Concrete Syntax & Abstract Syntax

a + b * c

Concrete = 3id and 1 plus and 1 multip

Abstract = What the compile will do (No ambiguity)

EBNF

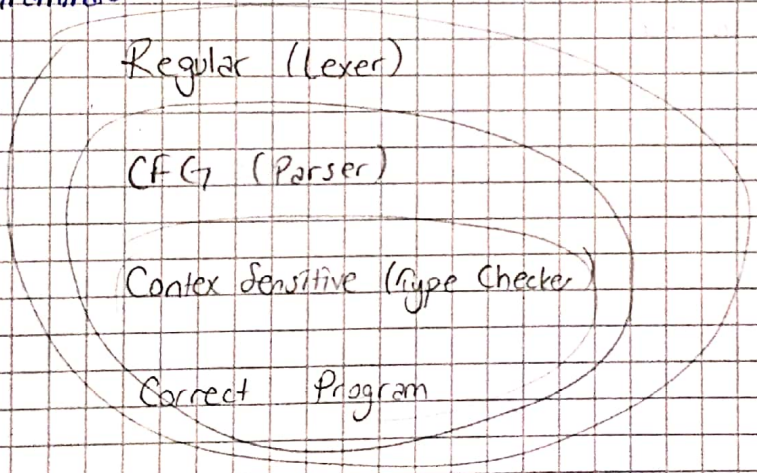
Extending BNF

for optinal part on RHS, use brackets

Use braces, for indefinite repeat or none

Multiple choice option → : term (*//+-+) factor

Grammars



Subject :

Date :...../...../.....

CHAPTER 3: VARIABLES

Variables, Binding, Scopes

- Variables
- Initialization
- Constants
- Binding
- Type checking
- Scope

Imperative languages are abstractions of Von Ne.

- Memory / Storage
- Processor / CPU

Variable: An abstraction of a computer memory cell or collection of cells.

Their design must consider

- Location
- Referencing
- Scope
- Lifetime
- Type checking
- Initialization
- Type Compatibility

Variable Attributes

- Name (usually)
- Address (In memory)
- Value (if initialized)
- Type (Range of values, interpretation, operations)
- Lifetime
- Scope

• Name: Not All VARIABLES ☺

Subject :

Date : ...../...../.....

Names

Set of characters
Names, or identifiers are used for variables
Comments, boundaries, whitespaces are not NAME!

C, C++, Java case sensitive

Variable Addresses

Memory address associated with variable

Also called l-value.
Location may change during execution

Variable Types and Values

- Type determines
  * range of values
  * set of operations that are defined for the variable
  * interpretation of bit patterns
  * for floating points, determines the precision
- Value " contents of the memory location (r-value)

The Concept of Binding

- l-value = address
- r-value = value

- Binding = Association
- Binding time

Variable Initialization

Initial binding of a variable to a value

Often done with the declaration statement

- Java (int sum = 0) ...

Subject :

Date :...../...../.....

Possible Binding Time

- Language design time (Bind operator symbols to operations)
- Language implementation time (Floating point type to a representation)
- Compile time
- Load time (Static)
- Run time (Non-static local variable to a cell)

Static vs. Dynamic Binding

Static

First occurs before run time
Remains unchanged throughout program execution

Dynamic

First occurs during execution
Can change during execution

Type Bindings

* How is a variable's type specified?
  - Using type declaration, variable declaration
* When does the binding take place
  - If static, type specified by explicit or implicit declaration

explicit: String name; name:...
implicit: String name:...

Dynamic Binding

JS, PHP, Clisp

list = [2, 4.33, 6, 8]
list = 17.3

Advantage: Flexibility
Disadvantage: Speed penalty for type checking

Subject :

Date : ...../...../.....

Static Variables

- Bound memory cells before program execution begins

* Adv : Efficiency
  History-sensitive subprogram support

* Disadv : Less flexible

Stack Dynamic Variables

Allocated from the run-time stack
Deallocate after execution

Adv: Allows recursion
Conserves storage

Dadv: Overhead of allocation
Subprograms cannot be history sensitive
Indirect addressing (Long time)

2 - Way

1- Runtime Stack
2- Heap Stack

Subject :

Date :...../...../.....

CHAPTER 4: DATA TYPES

Scope

The range of statements where a variable is visible

Local variables are visible in the program unit where they are declared

Non-local " " " in a program unit but not declared there

Static - Lexical Scope

To connect a name to a variable, one must find the declaration

Shadowed Variables

Variables are hidden in part of code where there is a more immediate ancestor with the same name

Dynamic Scoping

Adv: Convenience (no need to pass variable)
Flexibility

Disadv: Poor readability (same name different non-local var.)
Reliability (Accessibility from multiple subprograms)
Longer access times

Variables Revisited

A descriptor is the collection of the attributes of a variable

- If static, all attributes required only at compile time
- If dynamic, part or all need to be maintained during exec.

Subject : 30/11 Date : ...../...../.....

Simple Data Types

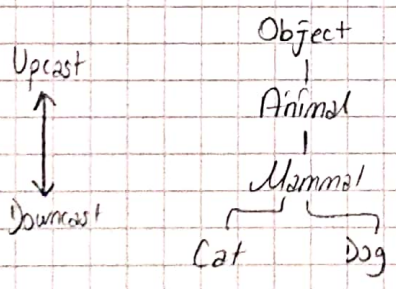
- No internal structure
  - * integers, double, char, bool
- Often directly supported in hardware
  - * Machine dependency
- Most predefined types
  - * String in Java.
- Some simple types are not predefined
  - * enum
  - * subrange

Type Conversion

Use code to designate conversion

No: Automatic / implicit
Yes: Manual / explicit

Upcasting / Downcasting



Subject :

Date : ../../....

Type System

- Type Constructor: Build new data types upon simple data types
- Type Checking: The translator checks if data types are used correctly
  - Type Inference: Infer the type of an expression, where data type is not given explicitly
  - Type Equivalence: Compare two types, if they are same
  - Type Compatibility: Can we use a value of type A in a place that expects type B

Nontrivial with user-defined types and synonyms types

Type Checking and Type Inference

- Standart Type Checking

```
int f(int x) {return x+1}
int g(int y) {return f(y+1)*2;}
```

Look body and use declared types of ids to check

- Type Inference

Look body without type information and figure out what types could have been declared.

```
f(x) = {___}
g(y) = {___}
```

Subject :

Date :...../...../.....

CHAPTER 5 : EXPRESSIONS

- Control : What gets executed, when, what order
- Abstraction Control
  - Expression
  - Statement
  - Exception Handling
  - Procedures and functions

Expression vs. Statement

Expression	Statement
No Side Effect	Side Effect
Return A Value (Comb. of variables) (2+10)	No Return A Value (Does something) (2 = ___)

Side effects : Changes to memory I/O / Can be undesirable (istenmeyen)

A program without side effects does nothing

Expression : No side effect, Order of evaluating sub expressions doesn't matter. (a+b)+(c+d)

Applicative Order Evaluation (Strict Evaluation)

Evaluate the operands first, then operator (bottom-up)

C ⇒ R to L
Java ⇒ L to R

Sequence Operator (expr1, expr2, ... exprn)

Non-Strict Evaluation

Evaluating an expression without necessarily evaluating all the sub expressions

- short-circuit Bool
- if - case

Subject : Date :...../...../.....

Normal Order Evalution

Operation evaluated before the operands

Operands evaluated only when necessary

2 (&) b
↓
first

- Normal Order (square (double 2))

double	x+x
square	x*x

1- square (double 2)
2- double (2) * double (2)
3- (2+2) * (2+2) L to R 4 * (2+2) → (4*4)=16

Applicative

1- square (double (2))
2- square (2+2)
3- square (4)
4- 4*4 = 16

CHAPTER 6: EXCEPTIONS

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space

Two main language constructs
- Declaration to establish excep. handle
- Statement or expression to raise or throw exception

Exception : It is an unusual event that is detectable by either hw or sw and that may require special processing

The special processing that may be required when an exception is detected is called exception handling. It is done by a code unit or segment called exception handler.

An exception is raised when its associated event occurs

Subject : Date : ...../...../.....

Adv: Without excep. , the code required to detect error condtitions.

It allow exception propagation. A single handler can be used in diff. location

It frees to programmer to consider all events

Simplify the code of programs that deal with unusual situations

Design Issues

Exception Binding

At the Unit Level: How can the same exception raised at differents points in the unit be bound to diff. handlers within unit?

At A Higher Level: If there is no exception handler local to unit, should exception be propagated to other units?

Continuation

Termination Simplest

Resumption ise useful when the condition encountered is unusual, but not erroneous.

Event Handling

Exceptions can be created eff. by user code or imp by hw or sw

Event are created by external actions, like GUI
Mouse clik etc

GARBAGE COLLECTION

1- Static Area (Fixed Size, Fixed Content, Allocated at Compile Time)
2- Run-Time Stack (Variable size, variable content)
3- Heap (Fixed Size, Variable Content)

Cell: Data item in the heap
Root: Register, Stack,

A cell is live if its address is held in a root

Subject :

Date : ...../...../.....

Garbage is a block of heap memory

Garbage Collection : Automatic management of dynamically allocated storage

Why GC?

- Memory leaks, dangling reference, double free

It is not a language feature

Perfect GC

- No visible impact on prog. exec.
- Works with any program and its data st.
- Collects garbage
- Excellent spatial locality of reference
- Manages the heap efficiency

Mark-Sweep GC

Each cell has a mark bit.

Heap dolana kadar uzemaz.

- Marking Phase

Start from root, set the mark bit on all live cells

- Sweeping Phase

Return all unmarked cells to the free list
Reset the mark bit on all marked cells

Adv: Handles cycles correctly
No space overhead

Dadv: Normal execution must be suspended
May touch all virtual memory pages
Heap may fragment (parcalanabilir)

Copy Collector

✓ - Very low cell allocation
✓ - Compacting

✗ - Twice the memory footprint

Subject : 42.02 Date :...../...../.....

CHAPTER 7: PROCEDURES

Procedure	vs.	Functions
Side effect, executed for it		No side effect
No return value		Return a Value
Procedure call: Statement		Function call: expression

Syntax

- Body
- Specification
  - × name
  - × type of return value
  - × parameters (names and types)

Caller to callee
Actual parameter to formal parameter

Parameter Passing Mechanisms

→ Call by Value → Normal / Pointers / Arrays

→ Call by Reference → formal parameters become alias of actual
Actual changed by changes of formal

→ Call by Value-Result

→ Call by Name → Actual used only need

→ Comb. of first 2 • Changes on at the end of block

Strict Evaluation

Applicative order
Call by val
" " ref
" " sharing
" " value-result

Non-Strict

Normal order
Call by name

Subject :

Date :...../...../.....

CHAPTER 8- ADT

DATA TYPES

- Predefined (int, string, ...)
- Type Constructors: Build new data types
- How to provide 'queue'
  - What should be the data values
  - What should be the operation
  - How to implement

Queue

Common ADT
FIFO

ADT

A mechanism of a pl design to imitate the abstract properties of a built-in type

Must include a specification of the operations

Must hide the implementation details

These properties are sometimes called encapsulation & inf hiding

Use of internal struc makes it difficult to change later

Operations on data not specified and often hard to hard

Encapsulation

All definitions of allowed operation for a data type in one place

Inf. Hiding

Seperation of implementation details from definitions. Hide details

Subject :

Date :...../...../.....

Algebraic Specification of ADT

Syntatic

The name of type, the prototype. 31:15

Semantic

Guide for required properties in implementation mathematical properties

They don't specify
- Data rep.
- Imp. details

type : queue(element) imports boolean

operations:
- create q : queue
- enqueue : queue x element → queue
- dequeue : queue → queue
- frontq : queue → element
- emptyq : queue → boolean

} except

Algebraic also

variables: q : queue x : element

axioms:

emptyq(createq) = true
emptyq(enqueue(q,x)) = false
frontq(createq) = error
frontq(enqueu(q,x)) = if emptyq(q) then x
else frontq

dequeu(createq) = error
dequeue(enqueu(q,x)) = if emptyq(q) then q
else enqueu(dequeue(q,x))

error axiom

Subject :

Date :...../...../.....

type Stack

ope createstck : stack
push : Stack x element → stack
pop : stack → stack
top : stack → element
empty : boolean

Constructor: Create, push

Inspector: Pop, top, empty

2x3 = 6 rules

s: stk x: elm

axioms:

emptystk (createstk) = true
" (push(s,x)) = false
top (create) = error
top (push (s,x)) = x
pop (createstk) = error
pop (push (s,x)) = s

Module: of program unit with a public interface and a private implementation; all services that are available from a module are described in its public interface and exported to the other modules

- Module can be (re)used in any way that its public interface allows
- " imp. can change without affecting the behavior of other modules

Modules are not types

" interface usually contains types, whose representations may be exposed (macro) (straw)

" are static

D.adv

- modules don't express semantic
- " don't expose dependencies

C++ namespace

Java packages

Subject : Date : ...../...../.....

CHAPTER 9 : PROLOG

Programmation in Logic

"What" instead of "How"

- Program

  Axioms (facts) : true statements

- Input to Prog

  query (goal) : state true or false?

Ex

Axioms : $0 \in N$
if x is a N, successor of x is N

Facts : Is $2 \in N$?
Is $-1 \in N$?

```
natural(0).
natural(N) : M is N-1,
             natural(M).
```

Factorial

factorial (0,X) ⇒ X=1

factorial (N,X) ⇒ NN = N-1
factorial (NN, X1)
X = X1 * N.

factorial (N) factorial (N,X)

```
factorial (0,1).
∀N and M,
factorial (N-1,M) → fac (N, N*M).
```

First Order Logic (FOL)

- Objects Things with individual identities (Students, lectures, companies)
- Properties of objects that distinguish (cayirmak) them from other (Blue, oval, large)
- Relations hold among sets of obj. (Brother-of, bigger than, outside)
- Functions which are a subset of relations where there is only "one" value or for any given "input" (father-of, Best friend)

Subject :

Date :...../...../.....

User Provides

- Constant (3, Green, Mary)
- Function father-Of (Mary) = John
- Predicate greater (5,3), green (Grass)

FOL Provides

- Variable Symbol x, y, foo
- Connectives : ¬ (not) → (implies)
  ∧ (and) ↔ (iff)
  ∨ (or)
- Quantifiers : Ax (universal) ∀x
  Ex (Existential) ∃x

BNF like that

S := <Sentence>
<Sentence> = ... blablabla

<Connective> = "AND", "OR" ...

- Every gardener likes the sun
  ∀x (gardener(x) → likesSun (x, Sun)
- All purple mushrooms are poisonous
  ∀x (mushroom(x) ∧ purple(x)) → poisonous(x)
- Clinton is not tall
  ¬tall (Clinton)

Declarative: Programs describe their desired results without explicitly listing commands

Subject : Date :...../...../.....

Horn Clause

$\forall x$ mammal(x) → legs(x,4) or legs(x,2)

Horn

legs(X,4) ← mammal(X) and not legs(X,2)

legs(X,2) ← " " " " (X,4)

C7C1)

gcd(U,0,U).

gcd(U,V,W) ← not zero(V), gcd(V, U mod V, W).

---

printpieces(L) :- append(X,Y,L)
write X
write Y
nl.
fail

[1,2]

[], [1,2]
[1], [2]
[1,2], []
no

CHAPTER 10: FUNCTIONAL PROGRAMMING

0- Logic P. = Axioms + Queries
1- Imperative Programming " Program = Data + Algo
2- OO " Program = Object.message
3- Functional " " = Functions functions

Functional P.L

Everything is function.

No variables or assignments

No loops (only recursive)

No side effect

First class values (Functions are values, can be parameters and return values)

Subject : Date : ...../...../.....

Adv : Simple semantics, concise (kısa, öz), flexible
NO side effect
less bugs

D.adv : Execution efficiency
More abstract and math., thus more difficult to learn and use

---

Lisp Scheme

Syntax
expression → atom | list
atom → number | string | id | char | bool
list → '(' exp seq. ')'

2 basic kinds of exp.

- atoms : unstructured
- list : the only str.

Eager Evaluation (Applicative order) : First sub-exp, then exp

Imperative : Arguments evaluate before they passed to call func

let → Binding a list (Not assignment)

eval → Get evaluation back (eval expression)

cons → construct a list

car → first element (car '(1 2 3)) = 1

cdr → the tail, without head (cdr '(1 2 3)) = (2 3)

lambda → (lambda (x) (* x x))
(define f (lambda ...))
(define (f x) (* x x))

Higher-Order function

- Returns a function as its value
- Takes a function as a parameter
- Both

(define (compose f g)
((lambda (x) (f (g x)))

Subject :

Date : ... / ... / ...

CHAPTER II: OOP

Object as activation records

Simula language (first) : imp as activation records with static scope

Pure Dynamically-Typed OOL

Object imp and run time lookup

Class-based lang. (Smalltalk)

Prototype-based lang (Self, JS)

Statically-Typed

C++ static typing to eliminate search

C++ problems with C++ multiple inheritance

Java using interfaces to avoid multiple inheritance

Primary OOL Concepts

Dynamic lookup

Encapsulation

Inheritance → Reuse of implementation

Subtyping → If I have multiple objects, how are these related? (Allows extension of concepts)

Dynamic Lookup

Object → message (arg) (OOP) x → add(y)

operation (operands) (Conventional P.) add (x,y)

Overloading is resolved at compile

(Dynamic lookup) is a run time operation

⇓

Different code for diff objects.

Subject : Date :...../...../.....

Encapsulation

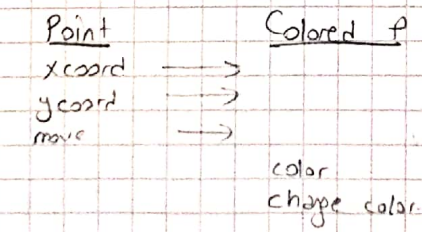
- User of a concept has "abstract" view
- Builder of a " " detailed "
- It seperates these two views
  - Imp. Code : Operate on representation
  - Client Code : Operate by applying fixed set of operations

Subtyping and Interface

- Interface : The external view of object
- Subtyping : Relation between interfaces
- Imp. : The internal rep. of an object
- Inheritance : Relation between imp.

- **Interface** The message understood by an object
  (change location example) → xcoord, ycoord, move

  The interface of an object is its type
- **Subtyping** If interface A contains all of interface B, then A obj. can also be used as B objects



Colored point interface contains Points

" " is a subtype of "

- **Inheritance** Imp. mechanism
  New objects may be defined by reusing imp of other obj.

Hoare claims that program design, doc, debugging are the most difficult aspects of the programming.

Subject : Date : ...../...../.....

Pure Dynamically-Typed OOL

Smalltalk (Major Language that popularized objects)

- Very flexible and powerful language
- Developed by Alan Kay
- Influence on Smalltalk
  - * Language intended to be pl and os interface
  - * Intended for "non-programmer"
  - * Syntax represented by language-specific editor

Terminology of Smalltalk

Object : Instance of subclass

Class : Defines behavior of objects

Selector : Name of message

Message : Selector together with parameter values

Method : Code used by a class to respond to msg

Instance var : Data stored in obj

Subclass : Class defined by giving incremental modifications to some subclass

Encap. of Smalltalk

Methods are public

Instance var. are hidden

- Not visible to other objects
- But may be manipulated by subclass methods

Subtyping : implicit, no static type system

Inheritance : Subclass, self, super

Subject :

Date :...../...../.....

- Self PL

Prototype-based pure OOL

Dynamically typed

Just in time compile

No classes (Everything is an object) (Everything done using msg)

No variables

The goal is conceptual economy and concreteness (somutluk)

JS Prototype

Object prototypes can be created using an object const. function

"new" to create object

Can add properties or methods to objects

× Statically Typed OOL

C++ Design Goals

- Provide OO features in C-based language
  - × Data abstraction
  - × Better static type checking
  - × Objects and Classes
- Important Principle

How Successful?

- Very well designed
- Many Users
- However, very complicated design
  - × Many features with complex interactions
  - × Difficult to predict from basic principles
  - × Most users chose a subset of language
  - × Many implementation-dependent properties

Subject :

Date :...../...../.....

Significant Constraints

- C has specific machine model
- No Garbage Collection
- Local Variables stored in activation records

C++ Object System

x Features

- Class
- Objects, with dynamic lookup of virtual functions
- Inheritance
  - Single and multiple inheritance
  - Public and private base classes
- Subtyping
- Encapsulation

x Some Good Decisions

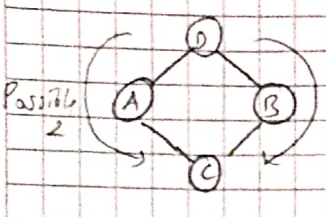
- Levels of visibility
- friend functions and classes
- Allow inheritance without subtyping
- Casts
- Lack of GC
- Object allocated on stack
- Overloading
- Multiple inheritance

1:26:39

Subject :

Date :...../...../.....

Diamond Inheritance



Possible 2

Possible 1

D members appears twice in C (Std Base class)

Avoid duplication (Virtual Base class)

JAVA LANGUAGE

- Oak language for "set-top box" (cable TV box etc.)
- Internet applications

Design Goals

- Portability (Bytecode interpreter on many platforms)
- Reliability (Typed source and language)
- Safety
- Simplicity and familiarity (Less complex than C++) (Everything is an obj)
- ~~Efficiency~~

Dynamic Lookup

Static Typing ⇒ More efficient than SmallTalk

Dynamic linking interfaces ⇒ Slower than C++

SON VIDEO ELEMMEDIA