5. Language Classification, Non-Structured Programming Languages

Aims of the Lecture

- Language categorization
  - Imperative languages
  - Declarative languages

What *Programming* Language

- *Program* may not describe computation
- Nevertheless, the program itself performs a computation
- Languages for
  - *General programming purposes*
  - *Special programming purposes*
  - *Type-setting*
  - *Etc.*

Data Abstraction Level

- Machine languages
- Higher level languages
  - Fortran, Cobol, Algol 60
- Universal languages
  - PL/I
- Structured languages
- Languages with abstract data types
- Object-oriented languages

Machine-Oriented Languages

- Data = sequences of bytes/bits; size matches format of machine instructions
- Operations:
  - Shift
  - Logic – AND, OR, XOR, etc.
  - Arithmetic – some of them in the beginning
- Higher logic data structures handled by program/programmer

Early Higher Level Languages

- Simple data types (HW abstraction)
- Strong orientation on target platform and kind of application – SC
- None of them a general purpose language

… this resulted in …

Language PL/I

- A large amount of data abstractions directly contained
  - Aspiration on general purpose language
- Creation of new abstractions not possible

Structured Languages

- Pascal, C, ...
- Simple constructs for definition of new data/control abstractions
- They follow program design methodologies
  - Searching a suitable abstraction
- Programs become more *readable*
  - Reflect problem structure, its solution

Languages with ADT

- Definition of data representation separated from applicable operations
- Data type representation/definition hidden from user (just declaration visible)
- Programs may be easier modified

OO Languages

- Data are tightly connected with operations that can manipulate them
- Data are in the center of developers focus, they are keys to the problem solution

Another Paradigms

- Logic languages
  - Predicates and terms as data, automatic proof
- Functional
  - Function as data
- Type setting
  - Font selection, text placement, etc.
- DML, DDL
  - Tables (SQL)
- ...

Abstraction of Control

- Imperative languages (procedurální)

- What a program *control* solves?
  - Which operations should be performed
  - In which order

Abstraction on the command level

- Machine languages
  - Low readability
  - Highly error-prone coding
  - Jump instruction
  - Slow development – grueling programming
- User oriented control structures
  - Usual design patterns: repetition, variant selection, ...

Abstraction on the Program Units Level

- Subroutines
  - Definitions, call/invocation, return, parameter passing
- Blocks
  - No name, thus, no explicit call
  - C, Algol 60
- Co-programs
  - Symmetric relation
  - Interleaved processing

Abstraction on the Program Units Level

- Parallel processing
  - Parallel run
  - Abstraction of CPU
  - Commands for synchronization
  - Mutual exclusion (competition)
  - Processes / thread
- Delayed processing
  - When the result is needed

Abstraction of Control II.

- Declarative languages
- What a program *control* solves?
  - Which operations should be performed
- The other task is solved by a compiler

Abstraction on the Program Units Level

- Not on other levels usually
- Similar as for imperative languages
- Order selection – various strategies (*functional, logic)*
    - *Call-by-need / lazy*
    - *Call-by-name / non-strict*
    - *Call-by-value / strict*

Terms to Remember

- Imperative, declarative, logic, functional, procedural, ... language

Exercises/Motivation

- Classify the following languages from several viewpoints (in/cross groups):
  - C, C++, C#, Java, JavaScript
  - Python, ML, Haskell, Hope, LISP
  - PHP, C, Java, C#, Perl
  - Pascal, C, ObjectPascal, Objective C
  - SmallTalk, Java, Objective C

Aims of the Lecture

- Features of non-structured languages
- How to implement data types in BASIC – proposal
- Programming in non-structured languages

Typical Representatives

- Fortran
  - Scientific and technical computation
    - Turbines, etc.
- Basic
  - An old “school” language
- Script/batch languages of these days
  - *Usually extended – remember PHP?*
- …

Features We Face Even Today

- Unstructured program control
  - goto
  - cycles/loops via if-goto (or similar)
    - usually wrong programming habit
- Unstructured data
  - data detachment
    - usually wrong programming habit
  - direct usage of bit-fields

So…

- Can we manipulate them formally?
- What is peculiar on them?
- What we should be aware of?

Is There Any Formal Base?

- What a formal base is?
  - Formal mean (calculus, algebra, …) that can be used for description of all language constructs
- What is it for?
  - Proof of language properties (soundness, semantics, etc.)
  - Implementation guide

Formal Base – Non-Structured Languages

- Usually not defined/used
- Backward creation usually not possible
  - Language design quite long ago
  - Control structures allow very complicated evaluation path
- Usually not required
  - Special targeting of languages

Syntax Description

- Natural language – example of usage
- Rarely formal one – (E)BNF
- Syntax very simple
  - Immediately combined with semantics
- “Open/fixed” syntax rules
  - Number/position of parameters
    - *C language*
  - Loop statement header per one line

Example

- Fortran
  - OPEN(UNIT = 2, NAME=“file”)
    - UNIT=<number> – compulsory
    - Other parameters (up to 15) optional
- BASIC

  -
    ```
    FOR I=1 TO 20 STEP 2
    PRINT I
    NEXT I
    ```

Semantics Description

- Informal – almost in 100%
- Examples follow description of syntax and semantics
  - Not too complicated X hidden semantics
- Incremental description
  - Simple first
- “Library” functions/procedures with semantics modified by parameter format/value

Example

- PRINT I; 20
  - Prints value of variable I and value 20
- PRINT USR 23760
  - Runs program and prints the returned value

Data Abstraction

- Very simple or no abstractions at all
  - Numeric types
    - No distinction between floating point and integer
    - Change of type is done “automatically”
  - Characters
    - Usually missing
    - Equal to strings
  - Arrays
    - Character string is equal to array of characters

Data/Control Manipulation

- Just built-in operations
- New types cannot be defined
  - Even existing ones cannot be grouped
- New operations can be defined in a limited way (within single file)
  - Simple combination of existing (macro)
  - Open subroutines

Open Subroutines

- Given routine/algorithm is stored within main file/program
- Control flows goes “around” it until it is explicitly required to execute it
- Entry point to the routine is not defined, just exit point
  - Parameters/result cannot be defined as a part of the routine
- GOSUB – RETURN

Example

```
…
100        LET K=5
110        LET I=10
120 …
…
160        RETURN
…
1000       GOSUB 110
…
5000       GOSUB 100
```

Insertion

- Functions/operations/procedures contained in another file(s) – **scripts/batches**
- Whenever operation defined in another file is to be used it is *inserted*
  - Include – PHP (not C)
  - Inline
  - Execute
  - Call
  - ...

Example (Mateth)

- …
  ```
  title('Puvodni zavislost');
  plot(x,y);
  yy = x;
  long;
  for n=5:8, rad=n; poce=160; ...
  exec('odmoapr.mmm',0); ...
  vysl, pause, ...
  odch, pause, ...
  ```
  …

- Isn’t PHP the same?

Any Other Languages?

- Nasty tricks
  - Javascript
- Half way
  - PHP
- Correct behavior
  - Python
- Syntax differences
  - BATches

Features of Such Subroutines

- Parameters passed as a global data only
  - Danger in overwriting another data – no local variables
- No kind of recursion implicitly supported
  - In certain cases, it can be solved explicitly by a programmer
- Program structure may be difficult to understand
  - Cannot see operations
  - Complex structure on the textual level

Program Design

- Data processed by a program code stored in the actual place
- Data visible since the definition/declaration point (explicit removal)
- Design methodologies can be hardly applied
- It is “easy” to make an error
- Team work very difficult

(Dis)Advantages

- Built-in operations and ad hoc solutions make programming “easier/faster”
- Complex programs are hard to implement in practice
- As it is impossible to use ADT the programming work is more difficult
- Closed subroutines are really missing

Types and Their Processing

- Non-typed languages usually
- Type may change during execution
  - Just by assigning another value
- Implicit types
  - Denoted by name of the variable
    - BASIC – A$, A
    - FORTRAN – names starting with N, I…
- Atomic types mainly – easy to manipulate

Demonstration Case

- Design of data types storage for non-typed language
  - Let us assume the following types:
    - Int
    - Float
    - String (text)
    - List

Types in Non-Typed Languages

- A variable is of inter-changeable type, in general
- The kind of type is such “as we need”
- Memory for variable is allocated when a value is assigned to it for the first time
- Memory size is large enough to contain all required numeric representations
  - Int X Float X String X List

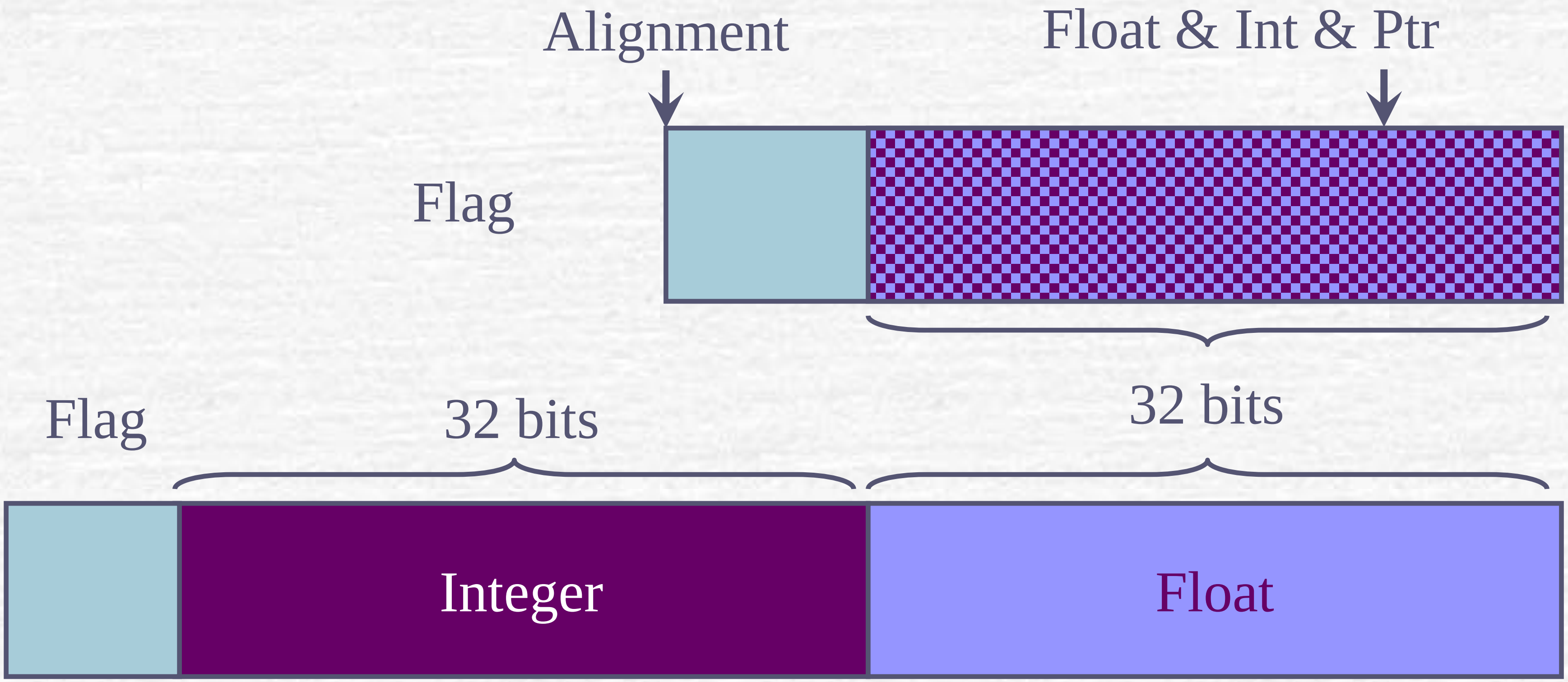
Storage

- Selection of encoding for numbers is very important task
  - Time of operation execution
  - Standards
    - Functions needed – libraries, HW
  - Detection of particular numeric type used
- For other types as well, but usually the possibilities not so wide

Not Suitable Design

- Let us take in account 32bit architecture with support for 64bit floating point numbers
- Possible errors
  - Wrong floating point format
    - e.g. proprietary, 32 bit when 64bit could be used
  - Structural/sequential storage
  - Storage requiring additional operation
  - etc.

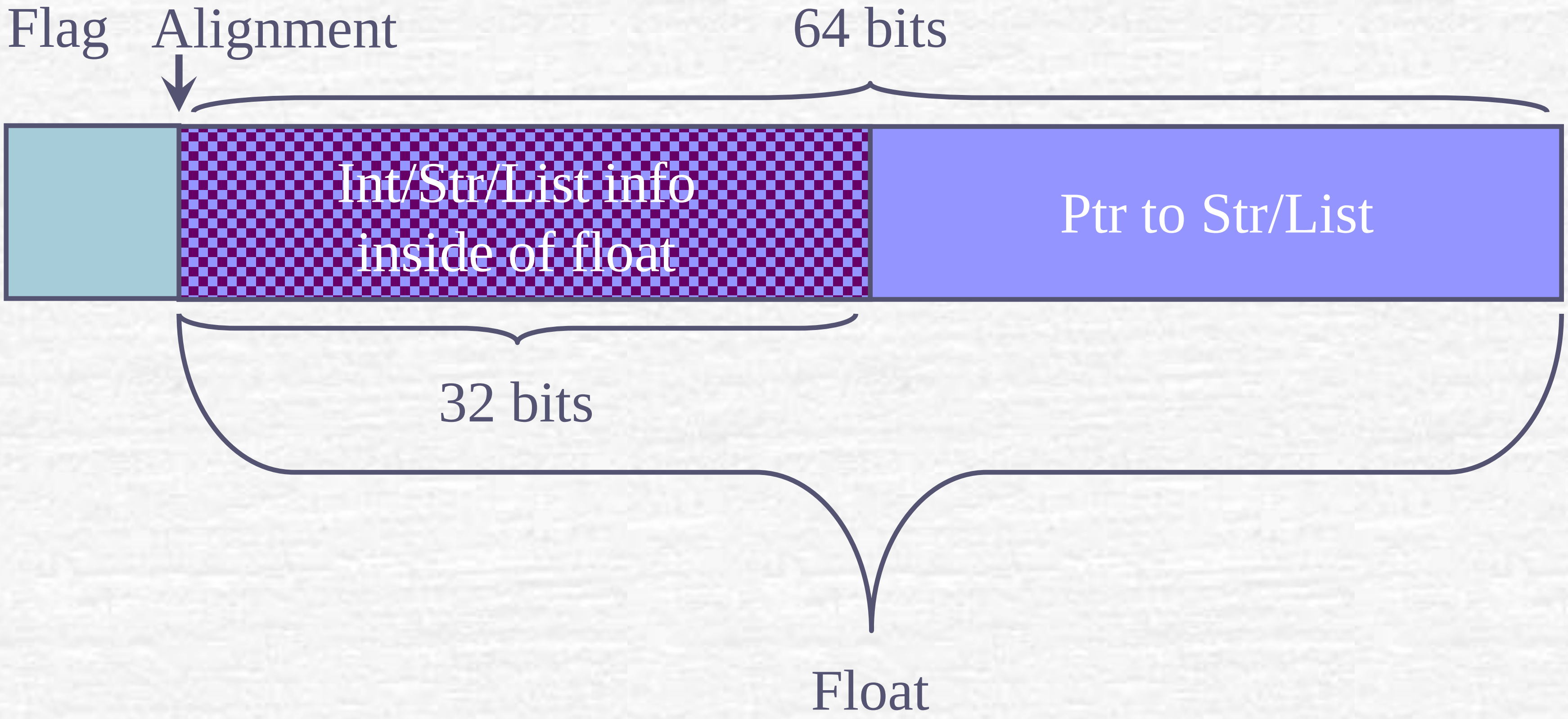
Not Suitable Design – Memory



A Better Design

- Exploit machine (CPU) features
- Overlap data types
- Align correctly
- Use tricks to
  - save memory
  - make evaluation faster

A Better Design – Memory



Evaluation During Run Time

- Simple situation
  - Operation for all type combinations
  - E.g. just two numerical types
- Type recognition
- Library/CPU function selection
  - E.g. 4 operations for subtraction(!)
- Operation execution
- Result storage

Evaluation During Run Time

- In the case, when the operation can be carried out even for more types the situation is even more complicated
  - Type conversion
  - Not-allowed combinations on input
  - Strategy of storage of types with variable length
  - De(Allocation) of memory

Arrays

- Usually declared before the first use
  - Number of array elements – in some languages
  - Memory allocation
- Strings (a kind of array)
  - Variable length
  - Upper bound on the length/size
  - Dynamic memory allocation – rarely used in practice (long strings)

Flow Control

- Control block-structured commands not supported (maybe loop FOR)
- Instructions of jump/conditional jump
  - IF + GO TO
- Subroutines not supported – just open ones
- Destination of jump instructions given by a row number, usually

Consequences

- Programs with highly complicated control flow may be created
  - Difficult or impossible optimization
- Difficulty of program design increases
- Checking during compile time limited
  - It is hard to detect when a variable is defined
- Conversion to structured design methodology (language) may be very complicated or impossible – F2C compilers

Formal V&V

- Program design V&V not used/possible in practice
- Floyd Hoare Logic can be applied only to some program constructions
- Primary usage of the languages by these days not in focus
  - History influence (Fortran)
  - Scripting – simple applets

Suitability for SE Methodologies

- All the presented features make it almost impossible to use SE methodologies
- Applicable only if programmers pay attention to guidelines and limitations
  - Hard to verify
- Algorithm abstraction – flow charts
  - Good for “nice” design
  - Language possibilities much broader though

Compiler Tasks

- Lexical level – no extra features
  - *Fortran – context analysis*
- Context-free features on the syntactic level
  - Used formal approach of formal languages and automata?
  - Context-free constructs
    - Loop FOR
    - Conditional evaluation IF-THEN(-ELSE)
    - Expressions (brackets)

Semantic Analysis

- We can safely detect during compilation
  - Existence of jump destination
- We can partially detect during compilation
  - Variable accessibility
  - Indices usage

Context Analysis

- One-level symbol table usually sufficient
- At the end of translation, it is known
  - Number of symbols used
  - Minimal size of memory allocation
    - Arrays
    - Strings
- Other kinds of context information cannot be mined

Code Generation

- Can be very simple
  - Calls to run time library
  - Flow control structured may make it more difficult
- Program processed as a whole
  - Suitable for high level of optimization (global)
    - Not applicable in general, though
      - Wrong design
      - Complex flow control
      - Etc.

Code Generation Result

- Not too efficient, but “readable” code
- Large run time libraries required
  - Automatically part of the result
- Optimization on the library level “only”
- CPU “exploitation” quite low
  - Many operations not directly involved in compuation

Interpreter Tasks

- Lexical analysis
  - Usual
  - During input from keyboard
    - Sinclair ZX Spectrum
    - IQ 151 ☺
- Syntactic analysis
  - Line-based translation
    - Various simplifications
    - “Intuitive” approach

Analysis of “New” Languages

- Line-oriented structure removed
  - Modern methods of analysis
- Pre-processed text not stored any more
  - Instant text analysis
  - Internal memory representation used
    - Abstract graphs/trees
    - Low-level code

Semantic Analysis

- May be performed in a sequence
  - Necessary information provided later
- During run time, all required issues can be solved
  - Not necessarily immediately
    - A “void” analysis is inserted
    - Limitation on a language level
      - Line length
      - Cycle end

Context Analysis

- One-level symbol table
  - Known symbols are contained in the table
- End of loop detection
  - Single line loops
  - Jump command “emulation”
  - New principles for newer languages
    - Stack utilization
    - No nested loops

Interpretation Itself

- Large run time library required
  - Quite similar to the one used for compiler
- Call to library functions not necessarily always
  - CPU features may be exploited – code in the interpreter design in-lined
- Evaluation performed always, when certain piece of program is successfully analyzed
  - Command, line, etc.

Recommendations/Warnings

- Not suitable for large projects
- If used for scripting then OK – scope and size is limited another way
- Not too much suitable for education
  - Various unsuitable programming habits
  - Good programming habits hard to achieve

Recommendations/Warnings

- If a language with such features used
  - Eliminate the feature, e.g. goto in C/C++
  - If not possible, use with care
  - Well document
  - *Change the language*
  - *Use framework*

Terms to Remember

- Types
  - Implicit
  - Type change during run time
  - Encoding of values
- Program storage
  - Compilation
  - Interpretation
- Open subroutines
- Data types, control flow

Exercises/Motivation

- Investigate your favorite scripting language and state:
  - What features are non-structured?
  - What features can be found in Java/C++?
  - What applications can be build with them?
  - What applications should not be build with them?
  - Would you use it to build a WWW site?