6. Block-Structured Languages

Aims of the Lecture

- Features of block-structured languages
- Pointers – usage/implementation
- Programming style in block-structured programming languages

Representatives

- Pascal (Wirth)
  - School language – usable for “paper” programming
  - *Pascal defined by prof. Wirth was not spread so much, nevertheless its modifications were!*
    - *Turbo Pascal → Free Pascal, Object Pascal → Delphi*
- Algol
- And others
- Structural features in many languages – PHP

Formal Base

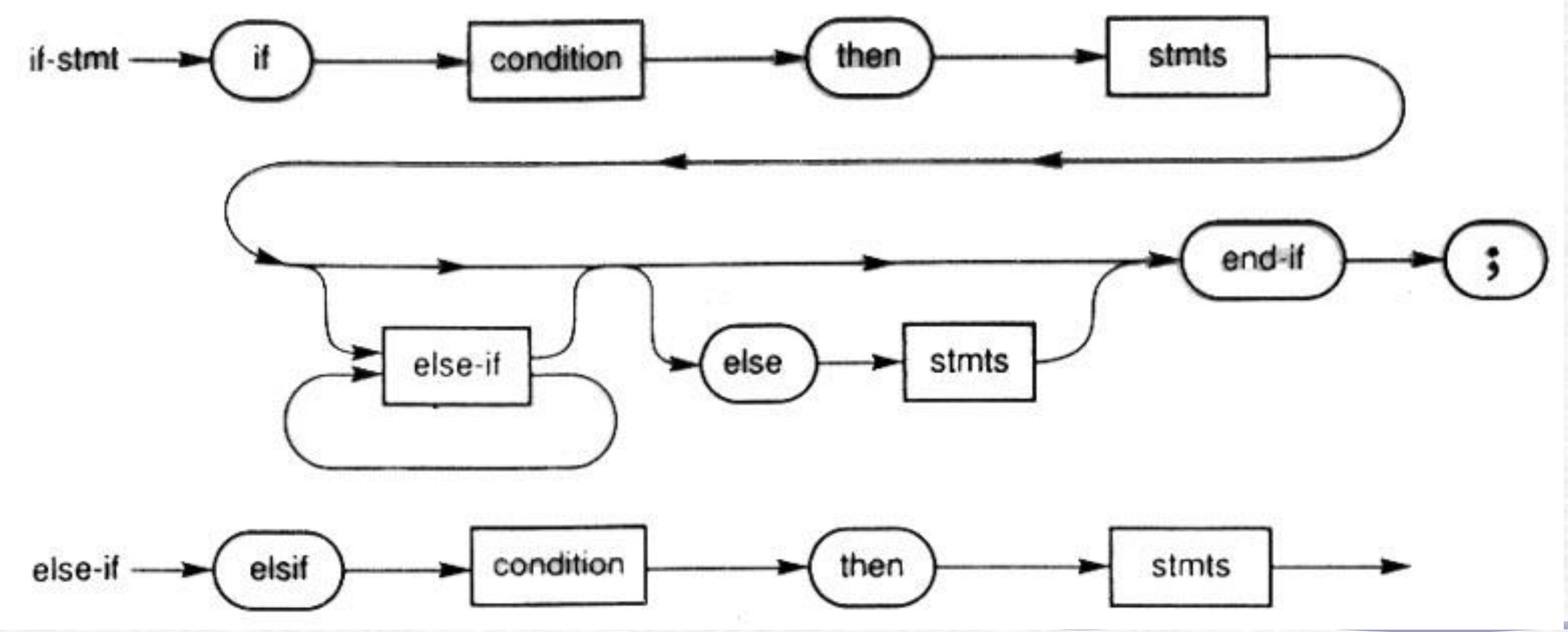
- In fact, not existing prior to language definition, nevertheless could be created
  - See references
- Language design based on experiences with languages used so far
  - Pros/cons taken in account
  - Language revisions – Algol (see language history)

Syntax Description

- Syntax already defined formally
  - Rich offer of commands, structures, …
- Together with textual information semiformal one
- Formal means used
  - Syntax graphs
  - (E)BNF
- CF languages/grammars

Example

```
<statement>
        ::= <unconditional statement>
        | if <expression> then <unconditional statement>
        | if <expression> then <unconditional statement>
                               else <statement>
```



Semantics Description

- Informal, usually
- Very thorough, though
  - Standards
- Not “explain by example”
  - Examples present, but their number is lower
- Sometimes even semiformal or formal explanations
- Quite large ☹
- *Compare with PHP manual!*

Example

- ALTER INDEX
- Purpose
  - Alters specific parameters for a spatial index or rebuilds a spatial index.
- Syntax
  - ALTER INDEX [schema.]index PARAMETERS (‘index_params [physical_storage_params]’);
- Keywords and Parameters
  - INDEX_PARAMS Allows you to change the characteristics of the spatial index, and the type (fixed or hybrid) of a quadtree index.
- Keyword/Description
- add_index
  - Specifies the name of the new index table to add. Data type is VARCHAR2.

3 pages in total

Data Abstraction

- Basic, atomic types
- Derived types – user defined, composition of other, already existing types
- Pointers or other mechanisms for definition of recursive data structures
  - Declarative programming
  - Lists, trees, etc.
- What can we do in PHP?

Data Manipulation

- The language usually contains operations for data access (read/write) for pre-defined types
  - Pervasive functions
  - Libraries – modularity required
- New operations for newly defined user types can be created using existing operations
  - Hierarchical nesting

Definition / Declaration

- User types must be defined/declared before the first use
  - Declaration limits usage
- Location of definitions/declarations
  - Strictly at the beginning of the program text
  - Any where (validity from the point of def.)
- Usage of declarations
  - Recursive data types via pointers
  - Mutually recursive functions

Program Design

- Structured languages allow to use at least some principles of “good” programming style
- Closed sub-routines can be used
  - Local variables
  - Data manipulation algorithms separated from main program flow
    - Repetitive usage
    - One place to modify/maintain
  - Name overlapping

Program Design

- Team co-operation on higher level
  - Closed sub-routines can be developed independently
    - Data type manipulation
    - Computational
    - Etc.
  - Program can be decomposed to isolated parts
    - Some may be reused
- One program file remains, though
- Data/name conflicts minimized
    - Global variables
    - Several authors

(Dis)Advantages

- “Libraries” can be created
  - Usually built in only (nothing new)
- Learning and full understanding is more difficult
- Program creation much more efficient
  - ADT can be used/created
  - Closed sub-routines
- Large programs can be created by several people
  - Rules must be obeyed, which may be treated as a limiting factor

Types and Their Processing

- Non-typed languages too, but in a minority
- Usually, explicitly typed entities
- Types of variables cannot migrate during evaluation
- Type conversions
  - Explicit X implicit
  - Solved during compile/analyze time
- User defined types increase complexity of a compiler/analyzer

Implicit Type Conversion

- Solved completely during compilation
  - ```
    x,y : real;
    x := y + 1024;
    ```
- During compilation decided on algorithm
  - ```
    x,y : real;
    i : integer;
    x := y * i;
    ```

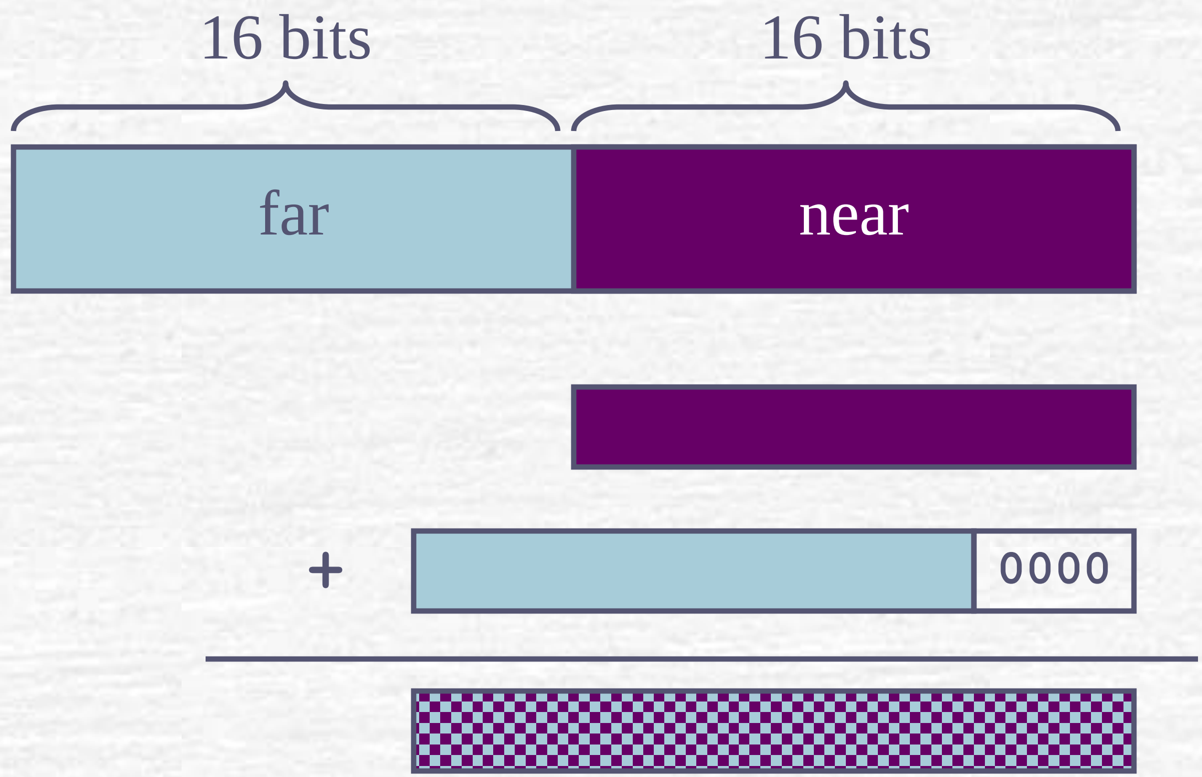
Explicit Type Conversion

- Conversion routine called (usually) – a *complex* algorithm
  - Round, floor, ceil, (int)3.1415926536…
- Sometimes can be compiled to simple operation on the target platform
  - ```
    int i; char c;
    c = (char)i;
    ```
    ```

Pointers

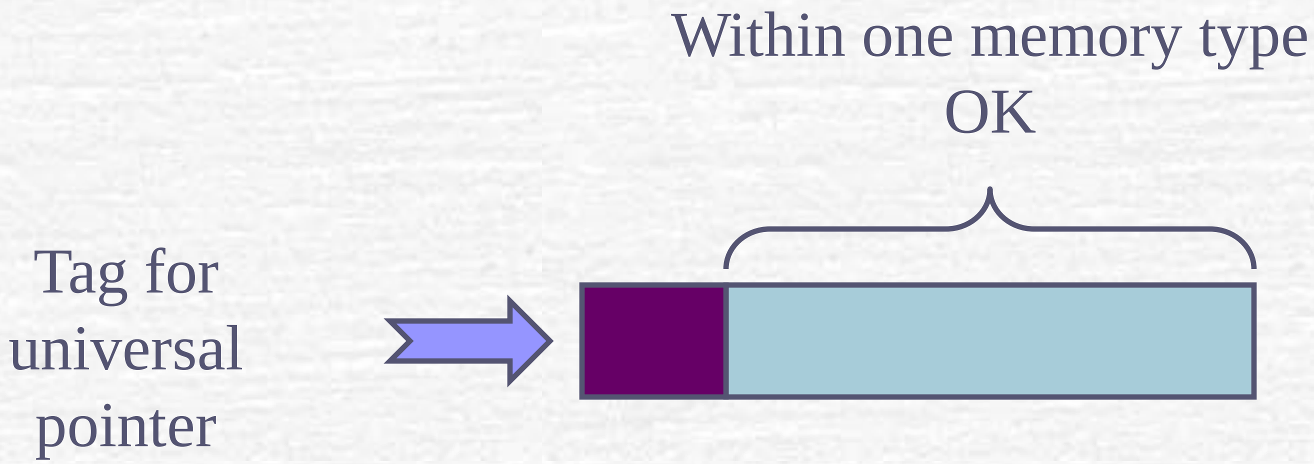
- Address of any(?) memory cell
  - Segmentation (application memory models)
  - Paging (usually user-transparent)
  - Data and program memory separated
- Varying memory and program complexity
  - Word x several words
  - CPU operation X sub-routine

Pointers – Memory Segmentation



Memory 1MB	20b
Segment 64kB	16b
No. of s. 64k	16b

Pointers – 2 Separated Memories



Harvard architecture

PHP – References

- Not only PHP ☺
- Access the same variable content by different names.
  - not like C pointers: e.g. no pointer arithmetic, not actual memory address, etc.
  - **symbol table aliases**
    - various names share the same data

PHP – Shared Content

- ```
  <?php
  $a =& $b;
  ?>
  ```
- It means that *$a* and *$b* point to the same content
- *$a* and *$b* are completely equal here
  - Not pointer of one to another
  ```

PHP – Pass by Reference

```
<?php
function foo(&$var) {
    $var++;
}
$a=5;
foo($a);
?>
```

PHP – Return Reference 1

```
<?php

  function &func() {
      static $static = 0;
      $static++;
      return $static;
  }
```

PHP – Return Reference 2

```
$var1 =& func();
echo "var1:", $var1;        // 1
func();
func();
echo "var1:", $var1;        // 3
```

PHP – Return Reference 3

```
$var2 = func();          // = without the &
echo "var2:", $var2;     // 4
func();
func();
echo "var1:", $var1;     // 6
echo "var2:", $var2;     // still 4
```

PHP – Return Reference 4

- Unlike parameter passing, here symbol & have to be used in both places
  - to indicate that one wants to return by reference, not a copy,
  - and to indicate that reference binding should be done, rather than usual assignment

Data Structures

- N-tuples (record, struct, …)
  - Name (of a type/variable) representing structure
  - Name/identification of an item
- Variants – overlapping structures (case, union, …)
  - Name
  - Identification
- Combination of previous

Storage in Memory

- N-tuples
  - One behind the other
  - Close binding (no gaps), though:
    - Alignment in memory
    - Memory access
    - Memory exploitation
    - Bit fields

Storage in Memory

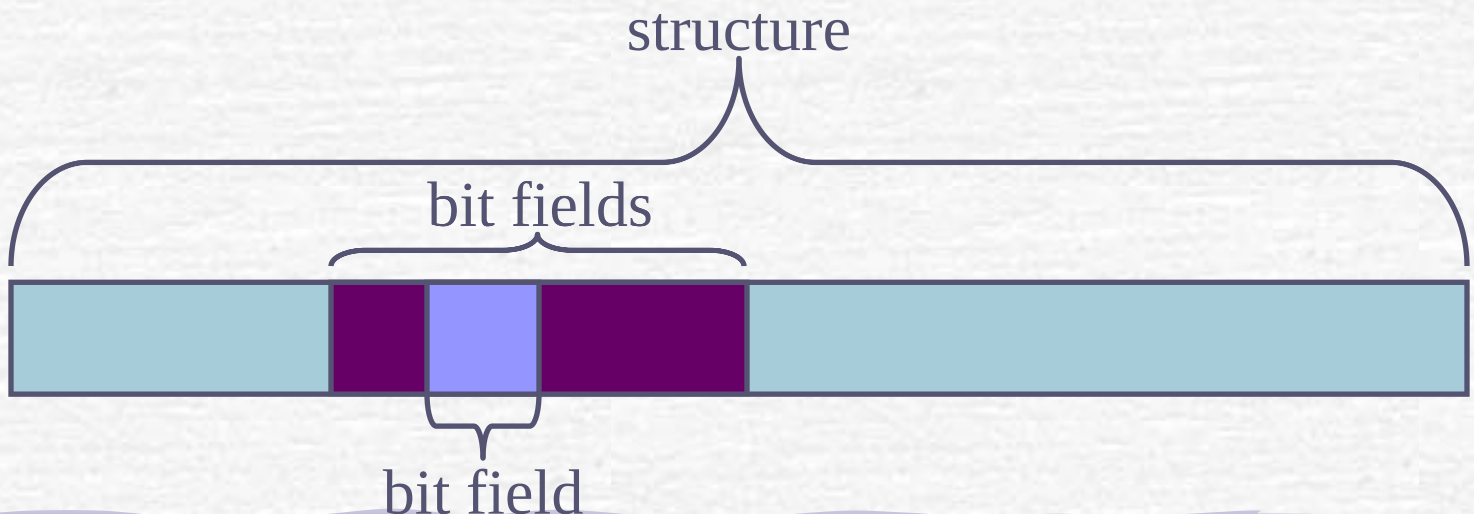
- Variants
  - Overlapping
  - Length is denoted by the largest item
  - Implementation tricks and hacks
  - Whole structure being aligned in memory
    - 2bytex X 4bytes

Manipulating Structures

- Pre-defined sub-routines X in-lined code
  - No support on the CPU level (special, optimized)
  - Item storage must enable “equal” access
- Item address
  - Denoted whenever the structure is accessed
  - Structure address
  - Offset
  - Static X dynamic evaluation

Manipulating Structures

- Bit fields
  - Base address and offset not sufficient, shift required
    - Other operations required – rotation/shift, mask



Arrays

- Storage depends on the language definition – usually elements directly one behind the other (no spaces/spaces)
- Problem – alignment of data structures
  - Alignment
    - Spaces to align
    - Item out of alignment
- Solution (language level/compiler level)
  - User level – packed array
  - Code generator
    - Complicated multiple access algorithm

Manipulating Arrays

- a: array [x..y] of <type>
  - Base address (constant)
  - a[i] = <base> + (i-x)*sizeof(<type>)
- Optimization
  - <base> + i*sizeof(<type>) - x*sizeof(<type>)
  - <base> - x*sizeof(<type>) + i*sizeof(<type>)

Constant (mapping function)

- Two dimensional arrays – similar way

Type Compatibility

- Array type/structure type
  - Name equality
  - Structure equality
    - Structure assignment
    - Structure as a function result
- Array as a unit

Passing Structures as Parameters Passed by Value

- Pointer
  - „Java"
- Copy
  - !! Extra large items inside (arrays, etc.)
    - Separate memory space, special manipulation
- Comes from the language definition
  - Approaches combined (C X Pascal X Java)

Structure as a Result

- Pointer (Java)
- Complete structure (C)
  - Register
  - Stack
  - Heap
- Not allowed – language definition (Pascal)

Flow Control

- Block creation/definition commands
  - begin end; { }
  - Sometimes local definitions on the block level
- Nesting of various constructions
  - Loops (break, continue)
  - Conditional commands
  - Multi-way commands
- Non-structured commands for flow control still present (go to)

Consequences

- “Nice” design allowed
- Old-fashioned manners can be used too, unfortunately
  - Advanced optimizing techniques X programs that cannot be optimized and *checked*
- Following certain rules, the program design can produce code that detects various errors during compile/analyze time

Formal Verification

- Certain constructions can be verified using formal constructs and approaches
  - Floyd-Hoare Logic
    - Rules for basic language constructions
    - Rules for numeric values and their equivalents

Floyd-Hoare Logic

- Pre- and Post- conditions defined for every
  - Command
  - Command sequence
  - Block
  - etc,.

Notation

- Let C denotes a command, P condition, which holds before command execution, and Q condition, which holds after command execution:
- $$\{ P \}\ C\ \{ Q \}$$
  - Partial correctness
- $$[P]\ C\ [Q]$$
  - Total correctness

Commands

- Assignment
  - V := E
    - V – variable
    - E – expression
- Sequence of commands
  - C_1; C_2; ... C_n;
    - C_i – command

Commands

- Block
  - BEGIN VAR V_1; V_2; … V_n; C END
    - V_i – variable
- Incomplete IF statement
  - IF S THEN C
    - S – logical expression
- Complete IF statement
  - IF S THEN C_1 ELSE C_2

Another Commands

- Loops
  - WHILE S DO C
  - FOR V:=E_1 UNTIL E_2 DO C
  - REPEAT C UNTIL S
- Etc.

Inference Rules

- Defined on the base of
  - Predicate logic
    - Log. extreme, proof, ...
  - Axioms of
    - F-H & predicate logic
  - Inference rules of
    - F-H & predicate logic

Demonstration of Inference Rule

- Rules for IF statements
- $$\frac{\vdash \{P \wedge S\}\ C\ \{Q\}, \quad \vdash P \wedge \neg S \Rightarrow Q}{\vdash \{P\}\ \text{IF } S \text{ THEN } C\ \{Q\}}$$
- $$\frac{\vdash \{P \wedge S\}\ C1\ \{Q\}, \quad \vdash \{P \wedge \neg S\}\ C2\ \{Q\}}{\vdash \{P\}\ \text{IF } S \text{ THEN } C1 \text{ ELSE } C2\ \{Q\}}$$

Example of a Proof

- $\vdash \{T \wedge (X \geq Y)\}$ MAX := X $\{MAX = max(X,Y)\}$
- $\vdash \{T \wedge \neg(X \geq Y)\}$ MAX := Y $\{MAX = max(X,Y)\}$

$$\Downarrow$$

- $\vdash \{T\}$
  IF X≥Y THEN MAX:=X ELSE MAX:=Y
  $\{MAX = max(X,Y)\}$

Rules for Inference Application

- Inferring from back to front
- Loop invariants play important role

Suitability for SE Methodologies

- Certain guidelines can be fully applied
  - Decomposition
- Unfortunately, limited
  - Team co-operation
  - Decomposition is limited
  - A possibility of unintended program modification still exists
- Speed and safety of program creation increases significantly

Compiler Tasks

- Starting with lexical up to semantic analysis, the operations can be easily chained within one pass
- Context-free languages/grammars, usually
- Context binding, of course
  - Symbol table(s)
  - Multi-level ones

Lexical Analysis

- More symbol categories than for non-structured languages
- As the program size increases, the most impact part of a compiler
- Semantically driven lexical analyzers
  - Change of the recognized symbol set
  - “Guess” of a category based on the previous/next symbol

Syntax Analysis

- Techniques for context-free languages analysis exploited
  - Predictive parsers (LL(1) – Pascal)
  - Bottom-up parsers (L(A)LR(1) – C)
  - Compiler constructors
- Multi-level symbol tables used
  - Several kinds of approaches
  - Name space separation
    - Kinds of entities (context influence)

Semantic Analysis

- Completely can be done
  - Type checking
  - Entity existence verification
  - Jump resolution
- Partially
  - Entity initialization
- Cannot be done
  - Wrong use of variables (overlapped) detection
  - Type conversion error detection

Interpreter Tasks

- Interpreters of such languages rare
  - “School”
  - Research
- Semantics is more complex
  - Block-end definition is “long-distant”
  - Jump targets below/beyond jump command

How to Do That?

- Analysis done on the block level
- Translation to internal representation
  - Code
  - Graph

How to Do That

- In general, program is executed in the 2nd pass or even later
  - Earlier error detection
  - Higher memory consumption
  - Higher speed of execution
    - No repetitive analysis
  - Difficult to use stop-n-change-n-go approach

Recommendations/Warnings

- It’s possible to use these languages even for “larger” projects
  - Not suitable for really large projects, though (especially by these days)
- Usually compilers exist
- Suitable for education
  - Some languages extended to become usable in practice
    - Modules
    - Objects

Recommendations/Warnings

- Usually remained on the school/research level
  - Exception – DELPHI (limitations can be observed anyway)
- When learned/studied deeply *professional* languages should be used instead

Terms to Remember

- Prototypes
- Type conversion
  - Implicit X Explicit
- Floyd-Hoare Logic
  - Inference rule
  - Proof
  - Loop invariant

Exercises/Motivation

- Use ANSI C to define ADT *polymorphic list* and operations over it (use recursion)
  - Is something similar possible even in Basic (not VB)?
  - If yes then how? (ADT, recursion)
- *Define in ANSI C an algorithm for evaluation of factorial (use loop) and try to prove its correctness in Floyd-Hoare Logic*