OPEN ABAL 4 CHATGPT

AMENESIK

Abstract

This document, describing the Open Abal language, was produced in conjunction with Chat GPT, in August 2025, after extensive discussion and interactive discussions.

Jamie Marshall

ijm@amenesik.com

Table of Contents

- Declaring Procedures (PROC)
- Parameters and Return Values
- Nested and Recursive Calls

7. **Input and Output**
   - Screen and Console I/O
   - Printing and Formatting
   - User Input
8. **Structured Data**
   - Fields and Buffers
   - Redefinition of Structures
   - Record Layouts
9. **Compilation and Execution Model**
   - Compilation Units
   - Linking
   - Execution Flow
10. **Error Handling**
    - THROW and Error Conditions
    - Error Handlers
    - Runtime Error Codes (e.g., pointer misuse)
11. **Program Modularity**
    - ATTACH and DETACH for Dynamic Program Linkage
    - Independent Execution Units
12. **Advanced Error Handling**
    - Nested Loops and BREAK
    - Structured Exception Flow
    - Errors 56/118: Pointer Misuse
13. **Dynamic Memory and Function Pointers**
    - CREATE for Memory and Function Binding

Chapter 1

Introduction to Open ABAL

Open ABAL (Advanced Business Application Language) is a high-level programming language designed for portability, robustness, and structured program development. Originally derived from BAL and early ABAL systems, it has evolved into a modern, binary-portable language that abstracts away the complexities of different operating systems, while preserving the performance and flexibility of compiled languages.

Open ABAL was created with the following goals in mind:

- **Binary portability**: A program compiled on one platform can be executed on another (e.g., Windows ↔ Linux) without modification.

- **Overlayed segments**: Memory efficiency is achieved through segmented program design. Only one segment is loaded into memory at a time, overlaid on the previous one.
- **Structured programming**: Encourages modular design using segments and procedures.
- **Native integration**: Seamlessly interfaces with external dynamic libraries (DLLs, SOs) through a portable mechanism.
- **Future-oriented design**: Conceived in the early 1990s to be distributed across networks, anticipating delivery via the future Internet.

1.1 Programs, Segments, and Procedures

An Open ABAL program is composed of **segments** and **procedures**.

- **Segments**: These are top-level program blocks. Only one segment is in memory at a time. Segment 0 is the entry point (similar to main in C). The object-oriented method keyword OVERLAY describes this mechanism.
- **Procedures**: Functions that provide modular logic within a program. They can define parameters and return values, and they support both global and local error handling.

```
PROGRAM "Example"
SEGMENT 0
  PRINT=1:”Program Start"
  LDGO.SEG 1
ESEG 0
SEGMENT 1
  CALL HELLOPROC()
  RET.SEG
ESEG 1
PROC HELLOPROC()
  DCL S$       ; Declare string variable
  S = "Hello, Open ABAL!"
  PRINT=1: S,TABV(1)
  EXIT
ENDPROC
END
```

In this example:

- The program begins in **Segment 0**.
- Control transfers to **Segment 1** with **LDGO.SEG**.
- **Segment 1** calls a procedure **HELLOPROC**, which prints a message.

1.2 Control Flow Instructions

Open ABAL provides several mechanisms for control flow:

- GOTO label → Transfers control unconditionally, without return.
- GOSUB label → Jumps to a label and expects a RETURN to come back.
- CALL procedure → Invokes a procedure, returns with EXIT.
- LDGO.SEG n → Loads and executes a segment, returns with RET.SEG.
- ON GOTO variable, label1, label2, ... → Branches to a label based on the value of variable.
- ON GOSUB → Same as above, but with return.
- ON TIMER interval GOTO/GOSUB label → Timer-controlled branching.

This combination allows structured and event-driven program designs.

Chapter 2 – Basic Program Structure

Open ABAL programs are structured in a clear and modular way, allowing developers to write portable, maintainable code while retaining low-level control over execution. This chapter introduces the essential program structure elements and explains how control is organized.

2.1 Program Entry Point

Every Open ABAL program begins execution in **Segment 0**, which serves the same role as the main function in C. Segments are the primary containers of program logic. Only one segment is loaded into memory at a time; when a new segment is loaded, it overlays the current one.

Example:

```
PROGRAM "Example"
SEGMENT 0
  PRINT=1:”Program started"
  LDGO.SEG 1
ESEG 0
SEGMENT 1
ESEG 1
END
```

Here, execution begins in Segment 0, prints a message, and transfers control to Segment 1 then terminates execution.

2.2 Segments

- **Definition**: A segment is a self-contained unit of execution.
- **Lifecycle**: Only one segment resides in memory at a time; loading another replaces the current one (overlay model).
- **Exit**: Segments terminate using the ESEG instruction, which behaves like STOP.
- **Transfer**: Control is transferred between segments using LDGO.SEG.

Example:

```
SEGMENT 1
  PRINT=1:”Now in Segment 1"
  RET.SEG
```

2.3 Procedures

Procedures are function-like constructs that provide reusable logic within a program. Unlike segments, they do not overlay but exist within the current memory scope.

- **Declaration**: A procedure is declared with the PROC keyword.
- **Termination**: Procedures exit with ENDPROC, which is equivalent to EXIT.
- **Parameters**: Formal parameters are declared inside parentheses.
- **Return Values**: Defined after a colon (:).

Example:

```
PROC add(x%, y%) : %
  DCL result%
  result = x + y
  RETURN result
ENDPROC
```

A call to this procedure:

```
DCL sum%
sum = CALL add(3, 5)
```

2.4 Control Transfers

Open ABAL supports several mechanisms to transfer execution flow:

- **GOTO label** – unconditional transfer, no return.
- **GOSUB label** – transfer with return via RETURN.
- **CALL procedure(...)** – invokes a procedure, returns with EXIT.
- **LDGO.SEG** – loads and executes a new segment, returns with RET.SEG.

2.5 Structured Loops

Open ABAL provides several loop constructs:

- **DO ... LOOP** – infinite loop (equivalent to while true in other languages).
- **REPEAT ... UNTIL (condition)** – executes until the condition becomes true.
- **WHILE (condition) ... ENDWHILE** – executes while the condition remains true.

Example:

```
DO
  PRINT=1:”Running forever"
  BREAK   ! escape nested loop if condition is met
LOOP
```

2.6 Exit Instructions

Depending on context, Open ABAL uses specific exit instructions:

- **BREAK** – leaves a nested loop.
- **EXIT** – leaves a procedure.
- **RET.SEG** – returns from a segment call.
- **RETURN** – returns from a GOSUB.
- **ESEG** – terminates the current segment (like STOP).

✅ **Summary of Chapter 2**:

- Execution begins at Segment 0.
- Segments overlay one another in memory.
- Procedures provide modular functions with parameters and return values.

- Multiple flow control instructions (GOTO, GOSUB, CALL, LDGO.SEG) allow structured or unstructured transfers.
- Loops can be infinite, conditional, or repeat-until.
- Exits are context-sensitive (BREAK, EXIT, RET.SEG, etc.).

Chapter 3 – Data Types and Declarations

Open ABAL provides a compact but powerful set of data types designed for system programming, database access, and communication with external systems. Unlike many modern languages, ABAL types are explicit in size and behaviour, allowing binary portability across platforms.

3.1 Declaration Syntax

All variables are declared using the DCL keyword. The declaration must specify:

1. **Name** of the variable.
2. **Size** in bytes (for scalar values).
3. **Optional dimensions** (for arrays).

Example:

```
DCL counter%=2      ; 2-byte integer
DCL message$=80     ; 80-byte string
DCL matrix:(10,10)  ; 2D array of 4-byte integers
```

3.2 Scalar Types

Open ABAL supports the following scalar categories:

- **Integers (%)**
    - Fixed size, declared with number of bytes.
    - Typically, 1, 2, 4, or 8 bytes.
- DCL small%=1
- DCL long%=8
- **BCD (Binary Coded Decimal)**

    - Used for financial and precise decimal calculations.
    - Declared with size in bytes.

- DCL balance=8 ! 8-byte BCD (16 decimal digits)
- **Strings ($)**
    - Contiguous memory of fixed length.
    - Null termination is not automatic.
- DCL name$=32
- **Pointers (PTR)**
    - Used to reference memory, structures, procedures, or external libraries.
    - Can be dynamically assigned.
- PTR buffer$=256

3.3 Arrays and Dimensions

Any type may be dimensioned into arrays. Dimensions are declared in parentheses after the variable name.

Example:

```
DCL table:(100)              ; array of 100 4-byte integers
DCL strings$(50,20)=32  ; 2D array, each string 32 bytes array
; access elements using parentheses:
table(5) = 42
PRINT=1: strings(3,10)
```

3.4 Structured Data (FIELD Redefinition)

Structured data is achieved by redefining memory areas using the FIELD=M,variable construct. This allows complex structures to be mapped onto raw memory.

Example:

```
PTR record$=72
FIELD=M,record$
  DCL name$=24
  DCL address$=48
FIELD=M
```

This declares a 72-byte record consisting of two fields: name$ and address$.

Nested redefinitions are possible, allowing structures within structures.

3.5 Constants

Constants can be defined using the CONST keyword. These are immutable values that can be used throughout the program.

```
CONST MAXLEN%=256
CONST PI=3.14159265358979
```

3.6 Aliasing

The ALIAS instruction allows one variable to point to the same memory as another, enabling reinterpretation of storage.

```
PTR buffer$=72
DCL aliasname$=72
aliasname = ALIAS(buffer)
```

This technique is often combined with FIELD=M for record interpretation.

3.7 Example: Structured Record

Here's a complete example of defining and using structured records:

```
PTR customer$=72
FIELD=M,customer$
   DCL name$=24
   DCL address$=48
FIELD=M
name = "John Smith"
address = "123 Example Road"
```

✅ **Summary of Chapter 3**:

- Variables are declared with DCL.
- Supported types: integers, BCD, strings, and pointers.
- Arrays may have one or more dimensions.
- Structures are built using FIELD=M.
- Constants are defined with CONST.
- ALIAS allows memory to be reinterpreted.

Chapter 4 – Expressions and Operators

Expressions in Open ABAL form the computational core of the language. They combine variables, constants, and operators to produce values. Open ABAL ensures explicitness in all cases, avoiding ambiguities that plagued early BAL/ABAL systems.

4.1 Assignment

Assignments use the = operator. The left-hand side must be a declared variable, while the right-hand side may be a constant, variable, or expression.

```
counter = 5
total = a + b
name = "OpenABAL"
```

Assignments operate with strict type compatibility. Conversion between types (e.g., integer to string) must be explicit.

4.2 Arithmetic Operators

Arithmetic applies to integer and BCD values.

Operator	Meaning	Example
+	Addition	a% + b%
-	Subtraction	a% - b%
*	Multiplication	a% * b%
/	Division	a% / b%
MOD	Modulo (remainder)	a% MOD b%

Example:

balance = balance + deposit

remainder% = counter% MOD 10

4.3 Relational Operators

Relational operators compare values and return a boolean condition (TRUE or FALSE).

Operator	Meaning	Example
=	Equal	a% = b%
!=	Not equal	a% != b%
<	Less than	a% < b%
<=	Less or equal	a% <= b%
>	Greater than	a% > b%
>=	Greater or equal	a% >= b%

These are mainly used in IF, WHILE, and REPEAT statements.

4.4 Logical Operators

Logical operators combine boolean expressions. **Parentheses are required to ensure unambiguous evaluation order.**

Operator	Meaning	Example
AND	Logical AND	(a% = 1) AND (b% = 2)
OR	Logical OR	(a% = 1) OR (b% = 2)
NOT	Logical NOT	NOT (a% = 0)

⚠️ **Important**:

- a = b OR c = b AND x = 1 is **ambiguous**.
- Always use explicit parentheses:

```
IF (((a = b) OR (c = b)) AND (x = 1))
   PRINT=1: "Condition met"
ENDIF
```

The compiler enforces warnings when parentheses are omitted.

4.5 String Operations

Strings in ABAL are fixed-length byte arrays. Assignments copy the entire content up to the declared size.

- **Concatenation**:
  Concatenation uses +.

- fullName = first + " " + last
- **Comparison**:
  Strings may be compared using relational operators.

```
IF name = "ADMIN"
   PRINT=1: "Access granted"
ENDIF
```

4.6 Operator Precedence

For historical compatibility, Open ABAL defines **no implicit precedence between "AND" and "OR"**.

Parentheses are **mandatory** for mixed logical expressions.

Arithmetic precedence is conventional:

1. Multiplication and division
2. Addition and subtraction
3. Parentheses override

Example:

```
result = (a + b) * (c - d)
```

4.7 Expressions in Control Flow

Expressions appear frequently in conditionals and loops:

```
IF ((balance > 0) AND (status != "CLOSED"))
   CALL ProcessAccount(balance)
ENDIF
WHILE (counter < 100)
   counter = counter + 1
WEND
```

4.8 Examples

Example 1: Numeric Operations

```
DCL a%=2, b%=3, c%=4
c = (a + b) * c
PRINT=1: c
```

Output: 20

Example 2: Logical Condition

```
IF ((userLevel = 1) OR (userLevel = 2)) AND (activeFlag = 1)
   PRINT=1:"User has access"
ENDIF
```

✅ **Summary of Chapter 4**:

- Assignments use = with strict type rules.
- Arithmetic, relational, logical, and string operators are supported.
- **Parentheses are required** for clarity in logical expressions.
- Strings are fixed length and may be concatenated or compared.
- Expressions are central to conditions and loops.

Chapter 5 – Control Structures

Control structures define the flow of execution in an Open ABAL program. They allow branching, looping, and structured error handling. The language enforces explicitness to ensure readability and portability across platforms.

5.1 Conditional Execution

The primary conditional construct is the IF statement.

5.1.1 IF...ENDIF

```
IF condition
   statements
ENDIF
```

Example:

```
IF balance > 0
   PRINT=1:”Account is active"
ENDIF
```

5.1.2 IF...ELSE

```
IF condition
   statements
ELSE
```

```
  alternativeStatements
ENDIF
```

Example:

```
IF userLevel% = 0
  PRINT=1:”Guest access"
ELSE
  PRINT=1:”Authenticated user"
ENDIF
```

5.1.3 IF...ELSE IF...ELSE ENDIF

```
IF condition1
  statements1
ELSE
IF condition2
  statements2
ELSE
  defaultStatements
ENDIF
ENDIF
```

Example:

```
IF userLevel = 0
  PRINT=1:”Guest"
ELSE
IF userLevel = 1
  PRINT=1:”User"
ELSE
IF userLevel = 2
  PRINT=1:”Administrator"
ELSE
  PRINT=1:”Unknown level"
ENDIF
ENDIF
ENDIF
```

5.2 Loops

Loops in Open ABAL are structured, and each requires explicit termination.

5.2.1 WHILE...WEND

Executes as long as the condition is true.

```
WHILE condition
  statements
WEND
```

Example:

```
counter = 0
WHILE counter < 10
  PRINT=1:counter
  counter = counter + 1
WEND
```

5.2.2 REPEAT...UNTIL

Executes at least once, then continues until the condition becomes true.

```
REPEAT
  statements
UNTIL condition
```

Example:

```
counter = 0
REPEAT
  PRINT=1:counter
  counter = counter + 1
UNTIL (counter  >= 10 )
```

5.2.3 FOR...NEXT

For counter-based iteration.

```
FOR variable = start TO end [STEP increment]
  statements
NEXT variable
```

Example:

```
FOR i = 1 TO 5
  PRINT=1:i
NEXT i
```

With a step:

```
FOR i = 10 TO 0 STEP -2
  PRINT=1: i
NEXT i
```

5.2.4 BREAK

BREAK exits a loop immediately. It may appear inside nested loops and always exits the **innermost loop**.

```
FOR i = 1 TO 10
  IF i = 5
    BREAK
  ENDIF
  PRINT=1:i
NEXT i
```

Output: 1 2 3 4

5.3 SELECT CASE

Structured multi-way branching is achieved with SELECT CASE.

```
SELECT expression
  CASE value1
    statements
  CASE value2
    statements
  CASE value3, value4
    statements
  DEFAULT
    defaultStatements
ENDSEL
```

Example:

```
SELECT day
  CASE 1
    PRINT=1: "Monday"
  CASE 2
    PRINT=1:  "Tuesday"
  CASE 3,4
    PRINT =1: "Midweek"
  DEFAULT
    PRINT=1:  "Weekend"
ENDSEL
```

5.4 Nested Structures

Control structures may be **nested** with no restrictions. Use of explicit iteration termination using ENDIF, WEND, etc., ensures readability.

```
FOR i = 1 TO 5
  IF MOD(I, 2) = 0
    PRINT=1: "Even number: " + CONV$(i)
```

```
  ELSE
    PRINT=1: "Odd number: " + CONV$(i)
  ENDIF
NEXT i
```

5.5 Structured Programming Principles

- No implicit GOTO jumps are allowed between structures.
- Each structure must be explicitly closed.
- BREAK is the only permitted early-exit keyword, ensuring clarity of intent.

✅ **Summary of Chapter 5**:

- IF, ELSE enable conditional branching.
- Loops include WHILE, REPEAT...UNTIL, and FOR...NEXT.
- BREAK provides controlled early exit.
- SELECT CASE allows clean multi-way branching.
- Explicit closures (ENDIF, WEND, etc.) enforce clarity and maintainability.

Chapter 6 – Procedures and Segments

Open ABAL supports **structured modular programming** through the use of *procedures* and *segments*.
These constructs allow the programmer to encapsulate functionality, promote code reuse, and support large-scale system development.

6.1 Procedures

A **procedure** is a block of code with a name, parameters, and optionally a return value. Procedures may be declared globally or locally within a segment.

6.1.1 Syntax

```
PROC procedureName (parameterList)
  statements
ENDPROC
```

6.1.2 Parameters

Parameters may be:

- **Value parameters** – copied into the procedure.
- **Reference parameters** – allowing modification of the caller's variable.

Parameter passing is determined by variable type and context.

Example:

```
PROC AddNumbers(a%, b%)
   EXIT (a + b)
ENDPROC

x = AddNumbers(10, 20)
PRINT=1:x          ; Output: 30
```

6.1.3 Returning Values

A procedure may return a value using EXIT.
This value may be used in subsequent assignments or expressions.

```
PROC Square(n%)
   EXIT (n * n)
ENDPROC

result = Square(7)
```

6.1.4 Local Variables

Variables declared inside a procedure are local in scope and cannot be accessed outside.

```
PROC Example()
   DCL temp%
   temp=10
   PRINT=1:temp
ENDPROC
```

6.2 Segments

A **segment** is a logical grouping of variables, and instructions.

6.2.1 Syntax

```
SEGMENT SegmentName or Number
   Declarations
   Instructions
ESEG
```

Example:

```
SEGMENT MathUtilities
  Dcl a%,b%,c%,x%
  C = a * b
ESEG
```

6.3 Calling Procedures in Segments

Procedures called from segments are invoked by name.

```
x = Multiply(5, 6)   ; Call inside MathUtilities
```

6.4 Recursion

Open ABAL procedures may call themselves, allowing recursive algorithms.

Example: factorial calculation:

```
PROC Factorial(n%) : %
  IF n <= 1
    RETURN 1
  ELSE
    RETURN n * Factorial(n - 1)
  ENDIF
ENDPROC
```

6.5 Error Handling in Procedures

Error handling may be declared inside a procedure with:

```
ON LOCAL ERROR GOTO &label, variable
```

This handler overrides any global error trap until the procedure ends.

Example:

```
PROC SafeDivide(a%, b%)
  Dcl err%
  ON LOCAL ERROR GOTO &handle, err
  RETURN a / b
&handle
  PRINT=1:"Division error, code="; err
  RETURN 0
ENDPROC
```

6.6 Segments and Modularity

Segments provide:

- **Namespace isolation**
- **Encapsulation** of related logic
- **Reuse** across multiple programs

A large project may consist of dozens of segments, each focused on a domain (e.g., Math, I/O, Database).

✅ **Summary of Chapter 6**:

- **Procedures** encapsulate reusable logic with parameters and return values.
- **Segments** provide overlay able application foundations.
- **Recursion** is supported.
- **Error handling** may be local to a procedure.
- **Segments** allow scalable, structured system design.

Chapter 7 – Data Types and Variables

Open ABAL provides a comprehensive set of data types to support business, scientific, and systems programming.

Variables are declared explicitly and can be **scalars, arrays, or pointers**.

7.1 Declaration of Variables

The DCL keyword is used to declare variables:

```
DCL variableName = size [ (dimensions) ]
```

- **variableName** – the identifier of the variable.
- **size** – size in bytes, BCD digits, or characters depending on type.
- **dimensions** – for arrays.

Example:

```
DCL age%          ; 2-byte integer
```

```
DCL name$=24     ; 24-character string
DCL balance=8    ; 8-byte BCD number
DCL matrix=4(10,10)  ; 10x10 array of 4-byte BCD numbers
```

7.2 Variable Naming Conventions

- Variables may include alphanumeric characters.
- Special suffixes indicate type conventions (historically adopted):
    - % for 16 bit integers
    - $ for strings
    - None for general numeric variables.

Example:

```
DCL productId%
DCL productName$=48
DCL price=8
```

7.3 Basic Types

Open ABAL supports the following basic types:

1. **Numeric Integers** – signed, size-defined (1, 2, 4, or 8 bytes).
2. **BCD Numbers** – decimal precision, commonly used for financial applications.
3. **Character Strings** – fixed-length character fields.

7.4 Arrays

Variables may be **dimensioned** with one or more indexes.

```
DCL sales=8(12)     ; Array of 12 8-byte BCD values
DCL grid%(10,10)    ; 2D array of 2-byte integers
```

Access by subscript:

```
sales(1) = 100.50
PRINT=1:sales(1)
```

7.5 Pointers

Pointers reference dynamically allocated memory. They are declared with PTR.

```
PTR buffer$=256
```

- Pointers may be **redefined** with FIELD=M to interpret memory differently.
- They may be allocated dynamically with CREATE.
- Released with REMOVE.
- Aliased to existing variables with ALIAS.

Example:

```
PTR data$=128
FIELD=M, data       ; redefine as structured fields
```

7.6 Constants

Constants may be defined using CONST:

```
CONST PI = 3.14159
CONST MAX_USERS = 1000
```

Constants cannot be altered at runtime.

7.7 Type Conversion

Explicit conversion is supported where this is meaningful.

```
DCL number=4
DCL text$=12

number = 1234
text = STRN(number)     ; Convert number to string
number = VAL(text)     ; Convert string back to number
```

7.8 Structured Buffers

A **structured buffer** is created by defining fields on top of a raw pointer:

```
PTR record$=72
FIELD=M, record
  DCL name$=24
  DCL address$=48
```

This allows structured access to database rows, file records, or network packets.

✅ **Summary of Chapter 7**:

- Variables are declared with DCL.

- Supported types: integers, BCD, strings, binary buffers, Booleans.
- Arrays are dimensioned with parentheses.
- Pointers enable dynamic memory, aliasing, and structured buffers.
- Constants are immutable.
- Conversion functions allow transformation between types.

Chapter 8 – Expressions and Operators

Expressions in Open ABAL combine variables, constants, and operators to produce values.
Operators follow strict precedence rules, and **explicit parentheses** must be used to avoid ambiguity.

8.1 Arithmetic Operators

Open ABAL supports standard arithmetic operations:

Operator	Meaning	Example	Result
+	Addition	A + B	Sum
-	Subtraction	A - B	Difference
*	Multiplication	A * B	Product
/	Division	A / B	Quotient
MOD	Modulo (remainder)	A MOD B	Remainder

Example:

```
DCL total=8

DCL price=8

DCL quantity=4


total = price * quantity
```

8.2 Relational Operators

Used in conditional expressions:

Operator	Meaning	Example
=	Equal to	A = B
<>	Not equal to	A <> B
<	Less than	A < B
<=	Less than or equal to	A <= B
>	Greater than	A > B
>=	Greater than or equal to	A >= B

8.3 Logical Operators

Logical expressions are **always round-braced** to avoid ambiguity.

Operator	Meaning	Example
AND	Logical conjunction	(A = B) AND (X = 1)
OR	Logical disjunction	(A = B) OR (C = B)
NOT	Negation	NOT (A = B)

⚠ **Important Rule**: Expressions such as:

```
IF A = B OR C = B AND X = 1
```

are **ambiguous**.
They **must** be written explicitly:

```
IF (((A = B) OR (C = B)) AND (X = 1))
```

This requirement stems from early BAL/ABAL design, ensuring consistent evaluation across compilers and runtimes.

8.4 String Operators

- **Concatenation**: +
- **Substring extraction**: SUBSTR(string, start, length)
- **Length**: LEN(string)

Example:

```
DCL firstName$=12
DCL lastName$=12
DCL fullName$=24

fullName = firstName + " " + lastName
```

8.5 Precedence Rules

Even though parentheses are **required in multi-clause logical expressions**, the general precedence rules are:

1. Parentheses ( )
2. Arithmetic * / MOD
3. Arithmetic + -
4. Relational = <> < <= > >=
5. Logical NOT
6. Logical AND
7. Logical OR

However, in **Open ABAL style**, programmers are encouraged to **always use parentheses** in conditional expressions.

8.6 Expression Evaluation Order

Expressions are evaluated **left-to-right** within the same precedence level.
Intermediate results may be stored and reused.

Example:

```
IF (((X + Y) > 10) AND ((A = B) OR (C = D)))
   PRINT=1: "Condition met"
ENDIF
```

✅ **Summary of Chapter 8**:

- Arithmetic, relational, and logical operators are provided.
- Logical expressions **must** be explicitly parenthesized.
- String operations support concatenation, substrings, and length checking.

- Precedence rules are defined but parentheses are enforced to eliminate ambiguity.

Chapter 9 – Control Structures

Control structures allow the programmer to guide the flow of execution within an Open ABAL program.

They include conditional branches, loops, and mechanisms to break out of nested structures.

9.1 Conditional Branching

9.1.1 IF ... ENDIF

The IF statement evaluates a logical condition and executes one or more instructions if true.

```
IF (A = B)
   PRINT=1:”Equal"
ENDIF
```

9.1.2 IF ... ELSE ... ENDIF

```
IF (X > 100)
   PRINT=1:”High value"
ELSE
   PRINT=1:”Normal value"
ENDIF
```

9.1.3 IF ... ELSEIF ... ELSE ... ENDIF

Multiple conditions may be chained:

```
IF (SCORE >= 90)
   PRINT=1:”Grade A"
ELSE
IF (SCORE >= 75)
   PRINT=1:”Grade B"
ELSE
   PRINT=1:”Grade C"
```

```
ENDIF
ENDIF
```

⚠ **All conditions must be explicitly parenthesized** to maintain clarity.

9.2 Loops

9.2.1 WHILE ... ENDWHILE

Executes instructions repeatedly while a condition remains true.

```
WHILE (COUNT < 10)
  PRINT COUNT
  COUNT = COUNT + 1
ENDWHILE
```

9.2.2 REPEAT ... UNTIL

Executes instructions **at least once** and repeats until the condition becomes true.

```
REPEAT
  INPUT "Enter a number: ", VALUE
UNTIL (VALUE = 0)
```

9.2.3 FOR ... NEXT

The FOR NEXT loop executes a block of code a fixed number of times, controlled by counters.

```
FOR I = 1 TO 10
  PRINT I
NEXT I
```

A STEP value may be added:

```
FOR I = 10 TO 1 STEP -1
  PRINT I
NEXT I
```

9.3 BREAK Statement

The **special label BREAK** is used to exit a nested loop (WHILE, FOR, REPEAT, or DO). Execution resumes immediately after the loop.

Example:

```
FOR I = 1 TO 100
  IF (I = 10)
```

```
    BREAK
  ENDIF
  PRINT I
NEXT I
```

9.4 NESTING of Control Structures

Control structures may be nested without restriction.
Clear indentation is recommended for readability.

Example:

```
FOR I = 1 TO 5
  WHILE (J < 10)
    IF ((A = B) AND (X > Y))
      BREAK
    ENDIF
    J = J + 1
  WEND
NEXT I
```

9.5 THROW for Error Signalling

The THROW statement allows a programmer to raise an error condition intentionally.
This integrates with the error-handling mechanisms described in **Chapter 12**.

```
IF (BALANCE < 0)
  THROW 201   ; Custom error code for "negative balance"
ENDIF
```

✅ **Summary of Chapter 9**:

- IF supports ELSE and ELSEIF clauses.
- Looping constructs include WHILE, REPEAT, FOR and DO.
- BREAK enables escape from nested loops.
- THROW raises error conditions, delegating handling to error traps.

Chapter 10 – Procedures, Segments, and Modular Structure

Open ABAL provides a structured way of organizing programs into **procedures** and **segments**.

These mechanisms allow modular development, code reuse, and separation of concerns, while preserving the binary portability of the language.

10.1 Procedures

A **procedure** is a reusable block of code that can accept parameters, perform computations, and optionally return a result.

10.1.1 Declaring a Procedure

```
PROC CALCULATE_SUM(A, B) : *
   EXIT( A + B )
ENDPROC
```

- PROC introduces the procedure.
- Typed parameters are listed in round brackets.
- The return type is specified after the : symbol.
- ENDPROC closes the definition.

10.1.2 Calling a Procedure

```
TOTAL = CALCULATE_SUM(5, 7)
PRINT=1:TOTAL
```

10.1.3 Nested Procedures

Procedures may call other procedures:

```
PROC DOUBLE(X) : *
   EXIT( X * 2 )
ENDPROC

PROC MAINPROC()
   DCL VALUE=8
   VALUE = DOUBLE(10)
   PRINT=1: VALUE
ENDPROC
```

10.2 Segments

A **segment** is similar to a procedure but does not return a value.
Segments are used for modular organization, initialization, or structured grouping of instructions.

10.2.1 Declaring a Segment

```
SEGMENT INITIALIZE
   PRINT=1:”System starting..."
ESEG
```

10.2.2 Invoking a Segment

LDGO.SEG INITIALIZE

10.3 Scope and Lifetime

- **Local variables** exist only within the procedure or segment.
- **Global variables** are declared outside and are accessible across program units.
- ON LOCAL ERROR handlers (see Chapter 12) apply only inside a given procedure or segment.

10.4 Parameters

Parameters may be:

- **By value** (copied into the procedure).
- **By reference** (shared pointer, especially with structured buffers).

Example:

```
PROC INCREMENT(VAR)
   VAR = VAR + 1
ENDPROC
```

10.5 Modularization with Segments

Programs may be divided into logical sections using multiple segments.
For example:

```
PROGRAM "BANKAPP"

SEGMENT MAIN
   LDGO.SEG LOGIN
   LDGO.SEG PROCESS_TRANSACTIONS
ESEG MAIN

SEGMENT LOGIN
```

```
  PRINT=1:"Enter password..."
ESEG LOGIN

SEGMENT PROCESS_TRANSACTIONS
  PRINT=1:"Transactions in progress..."
ESEG  PROCESS_TRANSACTIONS

SEGMENT 0
  LDGO.SEG MAIN
ESEG 0
END
```

10.6 Indirect Calls

Through function pointers (detailed in Chapter 13), both **procedures** and **segments** can be invoked indirectly using:

```
CALL (PROC_POINTER()) (PARAMS)
```

This allows highly dynamic program structures.

✅ **Summary of Chapter 10**:

- **Procedures** return values; **segments** do not.
- Procedures and segments support parameters and local scope.
- Modular structure improves maintainability and clarity.
- Indirect invocation is possible via function pointers.

Chapter 11 – Expressions and Logical Evaluation

Expressions in Open ABAL are combinations of **constants, variables, operators, and function calls** that yield a value.

To ensure **portability and predictability**, Open ABAL enforces strict rules for evaluation and requires explicit parenthesising of complex logical expressions.

11.1 Arithmetic Operators

The following arithmetic operators are available:

Operator	Meaning	Example
+	Addition	A = B + C
-	Subtraction	A = B - C
*	Multiplication	A = B * C
/	Division	A = B / C
MOD	Modulo (remainder)	A = MOD(B, 10)

⚠️ Note: % is **not valid** in ABAL; the keyword MOD must be used.

11.2 Relational Operators

Relational operators compare values and yield a logical result:

Operator	Meaning	Example
=	Equal to	IF (A = B)
!=	Not equal to	IF (A != B)
<	Less than	IF (A < B)
>	Greater than	IF (A > B)
<=	Less or equal	IF (A <= B)
>=	Greater or equal	IF (A >= B)

11.3 Logical Operators

Operator	Meaning	Example
AND	Logical conjunction	(A = B) AND (X = 1)
OR	Logical disjunction	(A = B) OR (C = B)
NOT	Logical negation	NOT (A = B)

11.4 Mandatory Parenthesising

Unlike some other languages, **ABAL does not allow ambiguous expressions**. Programmers must **always use parentheses** to indicate precedence when combining relational and logical operators.

Example of ambiguity:

```
IF A = B OR C = B AND X = 1
```

This would compile with a **warning** because it is ambiguous.

Correct forms:

```
IF ((A = B) OR (C = B)) AND (X = 1)
IF (A = B) OR ((C = B) AND (X = 1))
```

Both are valid but express **different logical requirements**.

11.5 Short-Circuit Evaluation

Logical expressions are evaluated **strictly left-to-right** as written inside parentheses.

- This means that values produced in earlier clauses can be reused in subsequent ones.
- Open ABAL does **not** apply automatic short-circuiting as in some languages; instead, explicit ordering via parentheses is required.

11.6 Arithmetic in Expressions

Expressions may freely combine arithmetic and logical operators, but explicit grouping is required:

```
IF ((MOD(A, 2) = 0) AND (B > 10))
   PRINT=1: "Even number greater than 10"
ENDIF
```

11.7 Warnings and Best Practices

- If parentheses are omitted, the compiler **issues a warning** and defaults to the **strict left-to-right evaluation order**.
- To maintain binary portability across systems, **always bracket logical subexpressions**.

✅ **Summary of Chapter 11**

- Arithmetic, relational, and logical operators are supported.
- MOD replaces %.
- **All logical expressions must be explicitly parenthesized**.
- Compiler enforces clarity by issuing warnings when ambiguity exists.

Chapter 12 – Error Handling and Exceptions

Error handling in Open ABAL provides mechanisms for detecting, trapping, and responding to exceptional conditions at both **global** and **local (procedure/segment)** levels.
This design ensures that programs remain robust and portable across systems.

12.1 Global Error Traps

The instruction

```
ON ERROR GOTO &label, variable
```

- Defines a **global error handler**.
- If any uncaught error occurs, control passes to &label.
- The variable receives the **error code**.

The trap remains active until explicitly Cancelled using:

```
ON ERROR ABORT
```

This cancels the global trap, restoring normal error propagation.

12.2 Local Error Traps

The instruction

```
ON LOCAL ERROR GOTO &label, variable
```

- Defines an error handler **local** to the current procedure or segment.
- If an error occurs while inside that scope, control transfers to &label.
- The variable receives the **error code**.

A local trap **masks the global trap** until:

- The procedure or segment exits, or
- The trap is cancelled explicitly with:

```
ON LOCAL ERROR ABORT
```

12.3 Raising Errors Manually

Open ABAL allows programs to **raise error conditions deliberately** using:

```
THROW value
```

Where value is a numeric error code (for example 56 or 118).
This is useful for:

- Signaling invalid states,
- Enforcing preconditions,
- Integrating user-defined validation with ABAL's built-in error system.

If an error handler is active, it will be invoked with the given error code.

12.4 Special Error Codes

Two common error codes when dealing with pointers are:

- **56** – Attempted operation on a pointer not allocated.
- **118** – Operation attempted on an inappropriate or invalid pointer type.

These are caught through the same error trapping mechanisms.

12.5 BREAK Statement

ABAL introduces a special label keyword:

```
BREAK
```

This provides an **immediate exit from a nested loop construct** (WHILE, REPEAT, DO).

Example:

```
WHILE (I < 100)
  IF (A = B)
    GOTO BREAK
  ENDIF
  I = I + 1
WEND
```

This form is clearer than deeply nested conditional exits.

12.6 Best Practices

- Use **global error traps** for application-wide safety nets.
- Use **local error traps** inside critical procedures for precise handling.
- Prefer **THROW** for user-defined or predictable error conditions.
- Always **clear traps** (ON ERROR ABORT, ON LOCAL ERROR ABORT) when leaving a critical section to prevent unintended handler persistence.

✅ Summary of Chapter 12

- ON ERROR GOTO defines global traps.
- ON LOCAL ERROR GOTO defines procedure-level traps.
- Local traps override global ones until explicitly aborted or scope exit.
- THROW raises custom errors.
- BREAK simplifies escaping nested loops.
- Error codes integrate both system and programmer-defined conditions.

Chapter 13 – Pointers, Memory Management, and Function References

Open ABAL provides powerful pointer and memory manipulation facilities that allow programs to dynamically manage data, redefine memory structures, and link procedures or libraries at runtime.

13.1 Declaring Pointers

A pointer is declared with PTR, specifying its size in bytes or as a structured object:

```
PTR BUFFER$ = 256      ; Defines a 256-byte pointer block
```

Pointers are first-class citizens in Open ABAL and may be redefined, aliased, or attached to programs.

13.2 Memory Redefinition with FIELD=M

The directive

```
FIELD=M,variable
```

allows redefinition of the memory area pointed to by a pointer.

- The redefinition is always **contiguous bytes**.
- It may be **nested** to any degree, allowing structured overlays.

Example:

```
PTR RECORD$ = 72
FIELD=M,RECORD
  NAME$ = 24
  ADDRESS$ = 48
FIELD=M
```

Here RECORD$ is redefined as NAME$ and ADDRESS$.

13.3 Memory Management Instructions

CREATE

Allocates memory dynamically for a pointer, optionally with dimensions:

```
CREATE POINTER$ (size [, dim1 [, dim2]])
```

- size = number of bytes per element.
- dim1, dim2 = optional dimensions for arrays.

Example:

```
CREATE TABLE$ (72, 100)   ; 100 rows of 72 bytes each
```

REMOVE

Releases previously allocated memory:

```
REMOVE POINTER
```

ALIAS

Makes a pointer reference another variable or pointer target:

```
POINTER = ALIAS(OTHER)
```

This does not copy memory; it simply makes both names refer to the same underlying area.

13.4 Program Attachment

One of ABAL's most innovative features is **dynamic program linkage**.

- **ATTACH** links a program dynamically to a pointer.
- **DETACH** disconnects it.

Syntax:

```
ATTACH POINTER (ProgramName)
DETACH POINTER
```

Example:

```
PROGRAM "example"
PTR EXAMPLE$ = #PTRSIZE * 2
FIELD=M,EXAMPLE
   PTR PROC METHODS(10)
FIELD=M
SEGMENT 0
ATTACH EXAMPLE (MYPROGRAM)
CALL (METHODS(1))()
DETACH EXAMPLE
ESEG 0
END
```

This mechanism directly inspired later concepts such as **Java class loading**.

13.5 Function Pointers

Open ABAL supports function pointers that may reference different kinds of executable entities:

```
DCL/PTR PROC name [dimensions] ; procedure pointer
DCL/PTR SEGM name [dimensions] ; segment pointer
DCL/PTR USER name [dimensions]  ; pointer to a dynamic (native) library function
DCL/PTR CODE name [dimensions] ; ambiguous, may point to any of the above
```

Example:

```
PTR PROC HANDLER
DCL PROC OTHER
```

13.6 Binding Function Pointers

The CREATE keyword connects function pointers to their targets:

CREATE HANDLER(PROCNAME) ! Bind pointer to procedure

CREATE FPTR(OTHERFPTR) ! Copy binding from another pointer

13.7 Indirect Invocation

Function pointers are invoked using CALL, with the pointer **round braced**:

```
CALL (HANDLER()) (params)
```

- The first parentheses indicate it is an indirect call.
- The second parentheses enclose the parameter list.

Example:

```
CALL (HANDLER()) (X, Y)
```

Return values may be used in expressions or assigned to variables:

```
RESULT = CALL (HANDLER()) (A, B)
```

13.8 Error Conditions

Errors in pointer operations (such as dereferencing an unallocated pointer or mismatched type usage) trigger system error codes:

- **56** – Pointer not allocated
- **118** – Invalid pointer usage

These must be trapped by the error-handling system described in **Chapter 12**.

13.9 Best Practices

- Always REMOVE dynamically allocated memory when no longer needed.
- Use ALIAS to avoid redundant data copying.
- Prefer ATTACH/DETACH for modular program structures.
- Always brace conditions and CALLs explicitly to avoid ambiguity.

✅ Summary of Chapter 13

- Pointers are dynamically allocated, redefined, aliased, and attached.
- FIELD=M redefines pointer memory into structured subfields.
- CREATE, REMOVE, and ALIAS manage memory lifecycles.
- ATTACH/DETACH enable program linkage.

- Function pointers (PROC, SEGM, USER, CODE) allow indirect calls.
- Errors (56, 118) are integrated into the global/local error handling framework.

Chapter 14 – Dynamic Libraries and External Interfaces

Open ABAL provides direct integration with external libraries, similar to Windows DLLs, Linux shared objects, or Java native libraries. This system allows ABAL programs to call into external binary code while maintaining **binary portability** across platforms.

14.1 Including Library Definitions

Before the PROGRAM keyword, external library definition files must be referenced using the #USER directive:

```
#USER "library.def"

PROGRAM "MAIN"
...
END
```

These .def files describe external functions that can be dynamically linked at runtime.

14.2 Structure of a Definition File

A definition file is a plain-text description of the library interface.

Example

```
VERSION = 1
RUNTIME = "example"
% MessageBoxA( $,$,$, % )
% GetTickCount()
END
```

Rules:

1. The file begins with a VERSION declaration.
2. The RUNTIME specifies the **library name** (not the platform-specific .dll or .so suffix).
    - Open ABAL constructs the actual filename at runtime.
    - This provides **binary portability** between Windows and Linux.

3. Each function is described by:
    - Return type (ABAL primitive type code)
    - Function name
    - Parameter list in round brackets (comma-separated ABAL types)
4. Parameters are **types only** (no names).
5. The file ends with the keyword END.

14.3 Supported ABAL Type Codes

External function definitions must use standard ABAL type codes:

- # – Integer 8-bit
- % – Integer 16-bit
- : – Integer 16-bit
- & – Integer 16-bit
- * – Packed decimal
- $ – string pointer

Example:

```
% CreateFileA( $, %, #, $, :, &, * )
```

14.4 Using External Functions

Once defined in the .def file, external functions become **first-class ABAL identifiers**.

They can be called directly:

```
MESSAGEBOXA(0, "Hello", "Caption", 0)
```

Or they can be accessed indirectly via function pointers:

```
DCL USER MSGBOX
CREATE MSGBOX(MessageBoxA)
CALL (MSGBOX()) (0, "Hello", "Caption", 0)
```

14.5 Dynamic Library Abstraction

Unlike traditional systems where the programmer must distinguish between DLLs and shared objects, Open ABAL abstracts away the differences:

- On **Windows**, RUNTIME="kernel32" maps to kernel32.dll.
- On **Linux**, RUNTIME="c" maps to libc.so.

This allows the same program to run **unchanged** across different platforms.

14.6 Error Handling in Library Calls

- If the library or function cannot be loaded, a runtime error is raised.
- Standard error trapping (see Chapter 12) must be used.
- Typical error codes include *invalid function reference* or *library not found*.

14.7 Example – Windows API

```
#USER "winapi.def"
PROGRAM "MAIN"
SEGMENT 0
   CALL MessageBoxA(0, "Open ABAL Example", "Demo", 0)
ESEG 0
END
```

With winapi.def containing:

```
VERSION = 1
RUNTIME = "user32"
INT MessageBoxA(PTR, PTR, PTR, INT)
END
```

14.8 Example – POSIX/Linux

```
#USER "posix.def"
PROGRAM "MAIN"
SEGMENT 0
   printf("Hello from Open ABAL on Linux!\n")
ESEG 0
END
```

With posix.def containing:

```
VERSION = 1
RUNTIME = "c"
% printf($)
END
```

14.9 Best Practices

- Always isolate external calls in a dedicated .def file.
- Use PTR for structures passed by reference.
- Ensure ABAL types are correctly mapped to the native ABI.
- Catch errors using ON ERROR GOTO around external calls.

✅ **Summary of Chapter 14**

- External libraries are integrated via #USER "file.def".
- Definition files use ABAL type codes only (no parameter names).
- Open ABAL abstracts DLLs vs. shared objects for binary portability.
- External functions become available as first-class ABAL functions or can be bound to function pointers.
- Errors are handled through the same structured mechanisms as internal errors.

Chapter 15 – File Handling and Database Integration

File handling in Open ABAL follows a **channel-based system**.
Each file (or database connection) is bound to a numbered channel, and all operations are expressed in the form:

```
INSTRUCTION = channel, input_parameters : label, error, output_parameters
```

This provides **clarity, portability, and structured error handling**.

15.1 File Types

Open ABAL supports four major file access methods:

- **(SQ)** – Sequential Block Files
- **(SI)** – Sequential Indexed Files
- **(MC)** – Multiple Indexed Files
- **(DB)** – Database Access

Each type has its own operations and usage model.

15.2 Opening and Assigning Files

Files are opened using the ASSIGN instruction:

```
ASSIGN = 1, "customers.dat", SI : NEXT, ERROR
```

- 1 – channel number
- "customers.dat" – file name
- SI – file type (sequential indexed)
- NEXT – label if successful
- ERROR – label if error occurs

15.3 Sequential Block Files (SQ)

Sequential block files are simple linear storage, suitable for logging or archival.

Writing Records

```
WRITE = 1: NEXT, ERROR, BUFFER
```

Reading Records

```
READ = 1 : NEXT, EOF, BUFFER
```

15.4 Sequential Indexed Files (SI)

Sequential indexed files allow indexed record retrieval by **primary key**.

Searching by Primary Index

```
SEARCH = 1, PRIMARY : FOUND, ERROR, BUFFER
```

Where:

- PRIMARY is the indexed key
- BUFFER is a structured data block to hold the record

Example Declaration

```
DCL NAME$ = 24
DCL ADDRESS$ = 48
DCL BALANCE = 8   ; BCD for money
PTR BUFFER$ = 80
FIELD = M, BUFFER$
DCL NAME$, ADDRESS$, BALANCE
```

Then:

```
SEARCH = 1, 12345 : NEXT, ERROR, BUFFER
```

Insert a New Record

```
INSERT = 1, PRIMARY: NEXT, ERROR, RECORD
```

Modify an Existing Record

```
MODIF = 1, PRIMARY : NEXT, ERROR, RECORD
```

Delete a Record

```
DELETE = 1, PRIMARY : NEXT, ERROR
```

15.5 Multiple Indexed Files (MC)

Multiple Indexed Files behave like Sequential Indexed but support **multiple secondary indexes**.

- **Primary index operations** (SEARCH, INSERT, MODIF, DELETE) are identical to SI.
- **Secondary indexes** are accessed using SQL-like queries via INXSSELECT.

15.6 SQL-like Access with INXSSELECT

To query data using secondary indexes, Open ABAL provides INXSSELECT.

Syntax

```
INXSSELECT = 1, "SELECT NAME, ADDRESS, BALANCE FROM CUSTOMERS WHERE
BALANCE > 0 ORDER BY NAME LIMIT 5" : NEXT, ERROR, ROWCOUNT, RESULTS
```

- 1 – channel
- SQL query – string
- ROWCOUNT – number of rows returned
- RESULTS – pointer variable to hold the result set

Managing Results

The pointer must be **altered** to match the result dimensions:

```
PTR RESULTS$ = (1)
INXSSELECT = 1, "SELECT NAME, ADDRESS, BALANCE FROM CUSTOMERS" : NEXT,
ERROR, ROWCOUNT, RESULTS
ALTER RESULTS(ROWLEN, ROWCOUNT)
```

Where ROWLEN is the total byte size of all columns (e.g., 24 + 48 + 8 = 80).

Each row can then be **aliased** to a buffer for column access:

```
FOR I = 1 TO ROWCOUNT
  BUFFER = ALIAS(RESULTS(I))
  PRINT=1: NAME, ADDRESS, BALANCE
NEXT I
```

15.7 Executing SQL Statements

For non-select operations, the INXSEXEC instruction is used:

```
INXSEXEC = 1, "UPDATE CUSTOMERS SET BALANCE = BALANCE + 100 WHERE
BALANCE < 0" : NEXT, ERROR
```

15.8 Database Integration (DB)

The DB access method allows direct access to external SQL database engines.
The interface is identical to MC file operations, with INXSSELECT and INXSEXEC providing query and command execution.

This abstraction ensures that an Open ABAL program can transparently switch between **indexed file storage** and **SQL database storage**.

15.9 Error Handling

All file operations use **structured error handling**:

- :NEXT → continuation on success
- :ERROR → label to handle failure (e.g., missing file, invalid index)
- Special conditions such as EOF or NOTFOUND may also be provided

Errors such as **invalid pointer operations** or **corrupted records** are reported with runtime error codes (see Chapter 12).

✅ Summary of Chapter 15

- Files are assigned by channel with ASSIGN.
- **(SQ)** sequential block – linear storage.
- **(SI)** sequential indexed – primary key management with SEARCH, INSERT, MODIF, DELETE.
- **(MC)** multiple indexed – SQL-like queries with INXSSELECT.
- **(DB)** – full SQL database access.

- All operations follow the INSTRUCTION = channel, ... : labels format.
