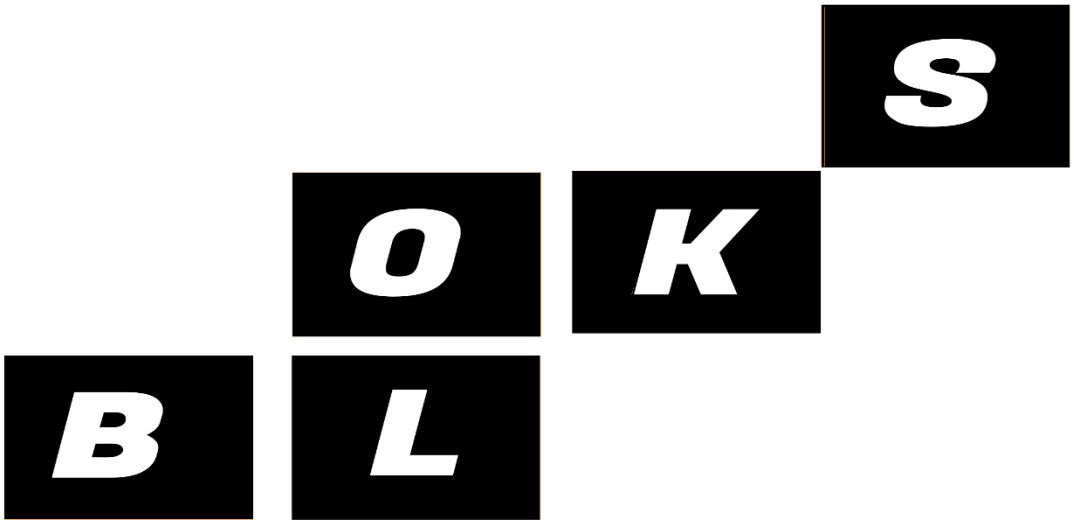


LANGUAGE REPORT

Version 1.0

February 2026

CONTENTS

1 Introduction

BLOKS is a text-based, block-structured programming language inspired by the Algol family, created to be readable, intuitive, and beginner-friendly. It was designed not as a modern computational language, but as a creative, educational, and playful tool for exploring programming concepts, scripting small programs, and building interactive stories.

Unlike traditional Algol-family languages, BLOKS emphasizes human-readable commands**,** flexibility**,** and experimentation. Its syntax is designed to resemble instructions or **"**building blocks**"** that can be stacked, nested, and reused, allowing learners and hobbyists to focus on logic and flow rather than strict compilation rules or complex syntax.

BLOKS differs from Algol in several distinctive ways:

- **English-like Commands:** Commands such as `say`, `set`, `repeat`, and `call` replace symbolic keywords and punctuation, making programs more approachable.
- **Dynamic Variables:** Variables are dynamically created and typed at runtime, unlike Algol's strictly typed declarations.
- **Readable Control Structures:** Loops and conditionals are expressed in plain language (`repeat N times … end`, `if X is Y … end`), improving clarity.
- **Block-Centric Design:** Every program is organized into **blocks**, which can be nested and called, allowing modular and hierarchical program flow.
- **Interactive and Playful:** BLOKS encourages experimentation with input, output, and branching stories, making it ideal for small interactive applications and learning exercises.
- **Interpreter-Based Execution:** Programs are run through an interpreter for immediate feedback, eliminating the compilation step common in Algol.

In essence, BLOKS transforms Algol's structured approach into a fun, educational, and highly readable programming experience, bridging the gap between learning programming and playing with code.

2 Fundamental Features of BLOKS

BLOKS is designed to be simple, readable, and interactive, providing an accessible way to explore programming concepts. While inspired by the Algol family, it intentionally avoids complex syntax and focuses on human-readable commands and block-based structure. This makes it ideal for beginners, hobbyists, and educational purposes.

2.1 Block-Based Structure

All BLOKS programs are composed of **blocks**, which are the core units of execution. Each block is defined as:

```
BLOK <block_name>
  <commands>
END
```

- Blocks can contain commands or other nested blocks, creating a hierarchical program structure.
- Blocks can be called from anywhere in the program using call <block_name>.
- This structure mirrors Algol's BEGIN…END blocks but in a more intuitive, readable format.

2.2 English-like Commands

BLOKS uses plain, human-readable commands instead of symbolic keywords and punctuation:

Command	Purpose
say "text"	Display text or variable values
set var to value	Assign values to variables
call block_name	Execute another block
repeat N times … end	Loop N times
if var is value … end	Conditional execution
input()	Read user input into a variable

This design allows programs to read like a series of instructions, making BLOKS easier to learn than traditional Algol-style languages.

2.3 Variables and Dynamic Typing

- Variables are **created automatically** on first assignment.
- They can store **numbers, strings, or input values**.
- Variables are **dynamically typed**, eliminating the need for explicit declarations.
- This flexibility contrasts with Algol's strictly typed variables, simplifying experimentation.

2.4 Input and Output

BLOKS provides straightforward I/O for interactive programs:

- **Output:** say "Hello" prints messages or variable values.
- **Input:** set choice to input() captures user input for decision-making.

These commands support interactive stories, games, and simple scripts.

2.5 Control Structures

- **Conditional statements:** if … is … end or if … is not … end.
- **Loops:** repeat N times … end, which can be nested indefinitely.
- **Block calls:** Use call <block_name> to structure execution flow modularly.

These structures are simpler and more readable than Algol's if…then…else and for…do loops, emphasizing clarity over complexity.

2.6 Playfulness and Educational Focus

BLOKS is designed to **encourage exploration**:

- Programs can include interactive choices, branching narratives, or mini-games.
- Immediate execution through the interpreter provides instant feedback.
- Focuses on logic and structure, allowing beginners to experiment without being slowed down by syntax errors.

3 Rules for Constructing BLOKS Programs

BLOKS programs are designed to be readable, modular, and easy to write. This section describes the basic rules for writing valid BLOKS code, including characters, identifiers, constants, and command construction.

3.1 Character Set

- BLOKS supports standard ASCII characters**.**
- Letters (`A-Z`, `a-z`), digits (`0-9`), spaces, and common punctuation are allowed.
- Commands and keywords are case-insensitive**,** so `say` and `SAY` are equivalent.

3.2 Identifiers, Constants, and Comments

3.2.1 Identifiers

- Names for variables or blocks must start with a letter and may include letters or digits.
- Examples: `main`, `playerHealth`, `choice1`

3.2.2 Constants

- Numeric constants: integers only (e.g., `5`, `42`)
- String constants: enclosed in double quotes (e.g., `"Hello world"`)
- Boolean values are represented through numbers (`0` for false, non-zero for true) or logical expressions

3.2.3 Comments

- Optional, written using `#` for single-line notes:

```
# This is a comment explaining the block
set health to 3
```

3.3 Basic Command Syntax

All commands must be indented under their block for readability, although the interpreter is whitespace-tolerant.

- **Say command:** outputs text or variable values

```
say "Welcome!"
say playerName
```

- **Set command:** assigns values to variables

```
set health to 3
set name to input()
```

- **Repeat loops:** repeat a fixed number of times or variable-driven

```
repeat 3 times
  say "Hello"
end
```

- **Conditional statements:** check values for flow control

```
if health is 0
  say "Game over!"
end
if choice is not "left"
  call right_path
end
```

- **Calling blocks:** modular execution of other blocks

```
call start
call fight_enemy
```

3.4 Nesting and Hierarchy

- Blocks can contain nested loops or conditionals, creating hierarchical structure.
- Commands inside a nested loop or conditional are indented to improve readability.
- Nested blocks can be called from anywhere, supporting reusable code.

Example of a nested structure:

```
BLOK main
  set counter to 2
  repeat counter times
    say "Outer loop"
    repeat 3 times
      say "Inner loop"
    end
  end
END
```

3.5 Execution Rules

1. Program execution **starts at** `BLOK main.`
2. Commands execute sequentially unless redirected by `call`, loops, or conditionals.
3. Variables are global to the program unless shadowed inside a block (advanced feature).
4. `end` terminates a block, loop, or conditional, ensuring clarity in structure.

3.6 Best Practices for Readable BLOKS Programs

- Keep blocks small and focused for clarity.
- Use descriptive names for variables and blocks.
- Indent nested commands consistently.

- Add **comments** to explain logic or purpose.

4 Program Structure

BLOKS programs are built around the block-centric design, where each block encapsulates a set of commands or nested blocks. Understanding the structure and execution flow of BLOKS is key to writing effective programs.

4.1 Blocks

- Every BLOKS program begins with **`BLOK <name>`** and ends with **`END`**.
- Blocks can contain:
  - **Commands** (say, set, repeat, etc.)
  - **Nested blocks**
  - **Conditional statements** and **loops**

Example:

```
BLOK main
  say "Starting program"
  call intro
END

BLOK intro
  say "Welcome to BLOKS!"
END
```

- Execution starts at **`main`.**
- Blocks act as self-contained units**,** making programs modular and reusable.

4.2 Nested Blocks and Calls

Blocks can be nested directly or via calls**:**

```
BLOK main
  call adventure
END

BLOK adventure
  repeat 2 times
    say "Step forward"
    call encounter
  end
END

BLOK encounter
  say "A wild creature appears!"
END
```

- `main` → `adventure` → `encounter`
- Loops and calls allow hierarchical and reusable flow.

4.3 Conditional Statements

- Syntax is simple and readable:

```
if health is 0
  say "Game over!"
end
if choice is "left"
  call left_path
end
```

- Supports **equality (`is`)** and **inequality (`is not`)** checks.
- Nested conditionals are allowed inside blocks and loops.

4.4 Loops and Repetition

- `repeat N times … end` repeats a block of commands.
- Loops can be **nested infinitely**, allowing complex sequences.

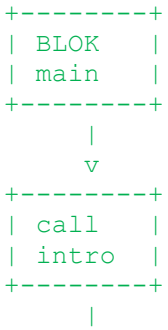
Example:

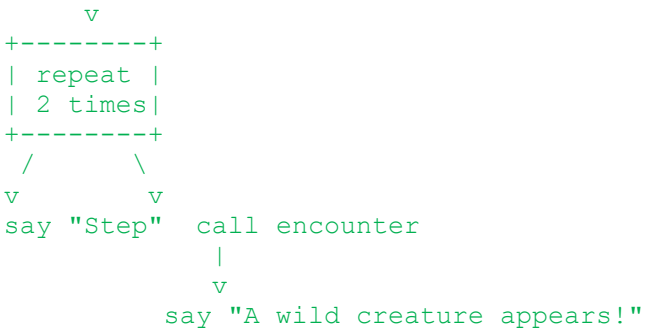
```
repeat 3 times
  say "Outer loop"
  repeat 2 times
    say "Inner loop"
  end
end
```

- This structure mirrors Algol's `for` loops but uses plain English, improving readability.

4.5 Execution Flow Diagram

Below is a **visual representation** of a typical BLOKS program flow:





- Arrows represent execution order**.**
- Loops and calls show hierarchical and modular flow**.**

4.6 Execution Rules

- Execution **always starts at `BLOK main`**.
- Commands inside a block execute **sequentially**.
- `call <block_name>` transfers execution to another block, then **returns** to the caller unless explicitly terminated.
- Loops and conditionals control **repetition and branching**.
- Variables are **global by default**, accessible across blocks.

4.7 Best Practices for BLOKS Program Structure

- Keep blocks short and focused for readability.
- Use descriptive names for blocks and variables.
- Indent commands inside loops, conditionals, and nested blocks.
- Comment complex logic to aid understanding.

5 Variables and Data Types

BLOKS uses dynamic and flexible variables to simplify programming and focus on logic and structure rather than strict type rules. This section describes how variables work, the types supported, and best practices for using them.

5.1 Declaration and Assignment

- Variables in BLOKS are **created automatically** on first assignment:

```
set health to 3
set playerName to "Aria"
```

- There is no need for explicit declarations; types are inferred at runtime.
- Variables can be reassigned freely:

```
set health to health - 1
set playerName to "Luna"
```

5.2 Numeric Variables

- Represent integers for counting, scoring, or looping.
- Can be used in arithmetic operations:

```
set score to 0
set score to score + 10
```

- Numeric variables can drive **repeat loops** or **conditional checks**.

5.3 String Variables

- Store **text values** for output, user input, or messages:

```
set name to "Shady"
say "Hello " + name
```

- Strings can be concatenated using + or manipulated in blocks.
- Used for interactive messages and branching stories**.**

5.4 Dynamic Variable Behavior

- **Type flexibility:** A variable can hold a number at one point and a string at another.

```
set temp to 5
set temp to "five"
```

- This eliminates the strict type constraints of Algol and allows rapid experimentation.

5.5 Boolean-like Behavior

- BLOKS uses numeric or logical expressions for conditions:

```
if health is 0
  say "Game over!"
end
```

- Non-zero numbers are treated as true, and `0` as false, simplifying conditional logic.

5.6 Arrays (Advanced)

- BLOKS can support **simple lists of values** (optional feature for extended learning):

```
set inventory to ["sword", "shield", "potion"]
say inventory[0]  # Outputs "sword"
```

- Arrays allow storing multiple related items and can be looped over in repeat statements.

5.7 References and Input Variables

- Variables can store **user input** directly:

```
set choice to input()
```

- Input values can then be used in **conditionals or loops**, making programs interactive.
- References between variables are **simple assignments**, allowing reuse and modularity.

5.8 Best Practices for BLOKS Variables

- Use descriptive names (e.g., `health`, `playerName`) for clarity.
- Initialize variables before use when possible to avoid unexpected behavior**.**
- Avoid mixing types in complex expressions unless intended; while allowed, it can confuse readers.
- Keep global variables minimal to maintain readability and modularity.

This section provides the foundation for managing data in BLOKS, showing how its dynamic, human-readable variables make learning programming easier and more intuitive than in traditional Algol-style languages.

6 Expressions and Assignments

Expressions and assignments in BLOKS are designed to be **clear, readable, and forgiving**, allowing programmers to focus on intent rather than syntax rules. BLOKS favors natural language over symbolic notation, making program logic easy to follow at a glance.

6.1 Expressions

An expression in BLOKS is any combination of values, variables, and operators that produces a result. Expressions are commonly used in assignments, conditionals, and loops.

Examples:

```
set total to score + 5
set remaining to health - 1
```

Expressions are evaluated **left to right** and use simple, familiar operators.

6.2 Arithmetic Operators

BLOKS supports basic arithmetic for numeric expressions:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

Example:

```
set points to points * 2
```

Arithmetic expressions can reference existing variables or literal values.

6.3 Assignment Statements

Assignments use the keyword `set` and follow a readable structure:

```
set variable to expression
```

Examples:

```
set lives to 3
set lives to lives - 1
set message to "You win!"
```

Variables are created automatically if they do not already exist.

6.4 String Expressions

Strings can be assigned directly or combined using the `+` operator:

```
set greeting to "Hello"
set greeting to greeting + " world"
say greeting
```

This allows simple text construction for messages and interaction.

6.5 Comparison Expressions

Comparison expressions are primarily used in conditional statements. BLOKS uses **word-based comparisons** for clarity:

- `is`
- `is not`

Examples:

```
if health is 0
  say "Game over"
end

if choice is not "left"
  say "Wrong path"
end
```

This differs from Algol-style symbolic comparisons and improves readability.

6.6 Logical Evaluation

Conditions in BLOKS rely on simple truth evaluation:

- `0` is treated as false
- Any non-zero value is treated as true

This keeps condition checks intuitive and easy to understand.

6.7 Nested and Compound Expressions

Expressions can reference other expressions through variables:

```
set damage to 2
set health to health - damage
```

While BLOKS avoids complex operator precedence rules, clarity is encouraged through **simple, step-by-step assignments**.

6.8 Design Philosophy

Unlike Algol, which emphasizes strict expression syntax and formal evaluation rules, BLOKS prioritizes:

- Readability over compactness
- Natural language over symbols
- Simplicity over optimization

This makes expressions and assignments in BLOKS ideal for learning, experimentation, and creative programming.

7 Control Structures

Control structures in BLOKS define how a program flows from one instruction to the next. They are designed to be clear, readable, and close to natural language, allowing programmers to express decisions and repetition without complex syntax.

7.1 Conditional Statements

BLOKS uses the `if` statement to make decisions based on conditions.

Basic form:

```
if condition
  commands
end
```

Conditions are written in plain language using comparisons such as `is` and `is not`.

Example:

```
if health is 0
  say "Game over"
end
```

Only the commands inside the block are executed when the condition evaluates to true.

7.2 Multiple Conditions

Multiple `if` blocks may be used sequentially to express different outcomes.

Example:

```
if choice is "left"
  call left_path
end

if choice is "right"
  call right_path
end
```

This approach keeps logic explicit and easy to follow, avoiding compact but hard to read constructs.

7.3 Repetition Statements

Repetition in BLOKS is handled using the `repeat` statement.

Basic form:

```
repeat number times
  commands
end
```

The number of repetitions may be a literal value or a variable.

Example:

```
repeat 3 times
  say "Hello"
end
```

This structure emphasizes intent rather than mechanics.

7.4 Nested Control Structures

Control structures may be nested inside one another, allowing complex behavior to be built from simple blocks.

Example:

```
repeat health times
  if health is 1
    say "Last chance"
  end
end
```

Nesting reinforces the block based design philosophy of BLOKS.

7.5 Block Based Flow Control

All control structures in BLOKS are explicitly closed using `end`. This avoids ambiguity and makes program structure visually clear.

There are no implicit scopes. Every block begins and ends clearly, reinforcing readability.

7.6 Comparison with Algol Control Structures

Compared to Algol, BLOKS control structures differ in several ways.

1. Keywords are written in natural language rather than symbolic or abbreviated forms
2. There is no requirement for parentheses around conditions
3. Blocks are always explicitly closed with `end`
4. Simplicity is favored over compact syntax

These differences make BLOKS easier to read and write, especially for beginners.

7.7 Design Goals

The goal of control structures in BLOKS is not performance or formal completeness, but clarity, approachability, and creativity. Programs should read like structured instructions rather than mathematical expressions.

8.1 The BLOK Structure

A block is defined using the `BLOK` keyword and terminated with `END`.

Basic form:

```
BLOK name
  commands
END
```

Each block represents a self-contained unit of behavior that can be executed from other blocks.

Example:

```
BLOK greet
  say "Hello"
END
```

8.2 The Main Block

Execution of a BLOKS program begins in a block named `main`.

Example:

```
BLOK main
  call greet
END
```

The `main` block serves as the program's entry point, similar to `main` procedures in Algol-derived languages.

8.3 Calling Blocks

Blocks are executed using the `call` statement.

Example:

```
call greet
```

When a block is called:

- Execution jumps to the named block
- The block's commands are executed sequentially
- Control returns to the calling block after completion

This makes program flow easy to trace and understand.

8.4 Block Reuse and Modularity

Blocks may be called multiple times from different locations in a program.

Example:

```
call greet
call greet
```

This encourages code reuse and reduces repetition, while maintaining readability.

8.5 Nested Block Definitions

Blocks may be defined anywhere in the program, but execution only occurs when they are explicitly called.

Nested blocks are treated as **logical structure**, not automatic execution units.

8.6 Parameters and Data Sharing

BLOKS blocks do not use formal parameters. Instead, they operate on **shared variables**, simplifying block interaction.

Example:

```
set score to 10
call update_score
```

This approach reduces complexity and aligns with BLOKS's educational focus.

8.7 Comparison with Algol Procedures

BLOKS blocks differ from Algol procedures in several ways:

- No formal parameter lists
- No explicit return values
- Shared dynamic variables instead of scoped declarations
- Plain language invocation using `call`

These differences favor clarity and experimentation over strict structure.

9 Program Execution Model

BLOKS programs are executed using an interpreter-based execution model, designed to provide immediate feedback and make program behavior easy to observe and understand. This model aligns with BLOKS's goal of being an exploratory and educational language.

9.1 Interpretation Instead of Compilation

BLOKS programs are not compiled into machine code. Instead, they are:

- Read line by line
- Parsed into blocks and commands
- Executed directly by the interpreter

This removes the need for a compilation step and allows programs to be modified and rerun quickly.

9.2 Parsing Phase

During parsing, the interpreter:

- Reads the source text
- Identifies `BLOK` definitions
- Groups commands into their corresponding blocks
- Builds an internal representation of the program

Blocks are stored by name and are only executed when explicitly called.

9.3 Execution Flow

Execution begins in the `main` block.

The interpreter processes commands sequentially:

1. Output commands (`say`) are executed immediately
2. Assignments (`set`) update variables
3. Control structures (`if`, `repeat`) alter execution flow
4. Block calls (`call`) temporarily transfer control

After a block finishes executing, control returns to the calling block.

9.4 Variable Handling During Execution

Variables are stored in a shared runtime environment:

- Variables are created on first assignment
- Values persist across block calls
- Changes made in one block are visible in others

This simplifies execution and avoids scope-related complexity.

9.5 Error Behavior

BLOKS favors simple and transparent error behavior**:**

- Unknown blocks result in runtime messages
- Undefined variables default to empty or zero values
- Syntax errors are handled at interpretation time

This approach encourages experimentation without fear of catastrophic failure.

9.6 Interactive Execution

The interpreter may pause execution to request user input:

```
set choice to input()
```

Execution resumes once input is provided, allowing interactive programs such as games and simulations.

9.7 Comparison with Algol Execution

Compared to Algol, BLOKS execution differs in key ways:

- No compilation stage
- Dynamic typing and runtime evaluation
- Immediate execution feedback
- Simplified error handling

These differences reinforce BLOKS as a learning and creative language rather than a production system.

10 Input and Output

Input and output in BLOKS are designed to be **simple, immediate, and human-centered**. Rather than abstract data streams or complex formatting rules, BLOKS focuses on direct interaction between the program and the user.

10.1 Output with `say`

The primary output statement in BLOKS is `say`.

Basic form:

```
say "text"
```

This prints text directly to the screen.

Examples:

```
say "Hello world"
say score
```

The `say` command can output:

- String literals
- Variable values
- Combined text and variables

This simplicity encourages experimentation and fast feedback.

10.2 Input with `input()`

User input is obtained using the `input()` function.

Example:

```
set choice to input()
```

Execution pauses until the user provides input, which is then stored as a string in the specified variable.

Input values may be used in conditionals, loops, or block calls.

10.3 Interactive Programs

Because input and output are straightforward, BLOKS is well suited for interactive applications such as:

- Text-based games
- Quizzes
- Simulations
- Educational demonstrations

Example:

```
say "Choose a path:"
set path to input()
```

10.4 Output Timing and Flow

Output appears immediately as commands are executed. There is no buffering or delayed display, making program behavior easy to observe and debug.

Optional extensions may include delays or time-based output for animations or live displays.

10.5 Comparison with Algol I/O

BLOKS input and output differ significantly from Algol-style languages:

- No device or channel specification
- No format strings or data stations
- Plain text interaction only
- Immediate execution feedback

11 Keywords

This section lists the reserved keywords used in the BLOKS language. Keywords have predefined meanings and cannot be used as variable or block names. The keyword set in BLOKS is intentionally small and readable, supporting the language's focus on clarity and approachability.

11.1 Block Definition and Execution

- `BLOK`
  Defines a named block of commands.
- `END`
  Terminates a block or control structure.
- `call`
  Executes another block.

11.2 Input and Output

- `say`
  Outputs text or variable values to the user.
- `input()`
  Reads a line of user input and returns it as a string.

11.3 Variables and Assignment

- `set`
  Assigns a value to a variable.
- `to`
  Connects variables to values in assignment statements.

Example:

```
set score to 10
```

11.4 Control Structures

- `if`
  Executes commands when a condition is true.
- `repeat`
  Repeats a block of commands a specified number of times.
- `times`
  Specifies repetition count in `repeat` statements.

Example:

```
repeat 3 times
  say "Hello"
end
```

11.5 Comparison Keywords

- `is`
  Tests equality between two values.
- `is not`
  Tests inequality between two values.

Example:

```
if health is 0
```

11.6 Logical Flow Keywords

- `main`
  Reserved name for the program entry block.

11.7 Keyword Design Philosophy

Unlike Algol-family languages with extensive keyword sets and symbolic operators, BLOKS uses plain English words with clear meaning. This makes programs readable even to users with little or no programming experience.

The keyword list is intentionally limited to reduce cognitive load and encourage experimentation.

12 Appendix

This appendix provides reference material, example programs, and licensing information for the BLOKS language. It is intended as a quick guide and supporting documentation for readers of the main report.

12.1 Quick Reference Cheat Sheet

Program Structure

```
BLOK main
  commands
END
```

Output

```
say "Hello world"
say variable
```

Input

```
set choice to input()
```

Variables

```
set score to 10
set name to "Alex"
```

Conditionals

```
if score is 0
  say "Game over"
end
```

Loops

```
repeat 5 times
  say "Hi"
end
```

Block Calls

```
call other_block
```

12.2 Example Programs

Hello World

```
BLOK main
  say "Hello world"
END
```

Simple Counter

```
BLOK main
  set count to 3
  repeat count times
    say count
    set count to count - 1
  end
END
```

Interactive Choice

```
BLOK main
  say "Left or Right?"
  set choice to input()
  if choice is "left"
    say "You chose left"
  end
  if choice is "right"
    say "You chose right"
  end
END
```

12.3 Formal Syntax Summary

This summary describes the basic grammar of BLOKS in an informal notation.

```
program      ::= block+
block        ::= "BLOK" identifier command* "END"
command      ::= say | set | if | repeat | call
say          ::= "say" expression
set          ::= "set" identifier "to" expression
if           ::= "if" condition command* "end"
repeat       ::= "repeat" expression "times" command* "end"
call         ::= "call" identifier
```

12.4 Design Goals Recap

BLOKS was designed with the following goals:

- High readability
- Minimal syntax
- Immediate feedback
- Educational and creative use
- Fun over formality

It is not intended to replace established languages, but to encourage exploration and learning.

12.5 GNU General Public License Disclaimer

BLOKS is free software. You may redistribute and modify it under the terms of the **GNU General Public License**, version 3 or any later version.

BLOKS is distributed in the hope that it will be useful, but **without any warranty**; without even the implied warranty of merchantability or fitness for a particular purpose.

A copy of the GNU General Public License should be included with this software. If not, see the Free Software Foundation for the full license text.