LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

# Specification & Implementation

Författare

**Mattias Ajander**, mataj513@student.liu.se

**Ludwig Moström**, ludmo578@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2025
Version 1.0

2025-02-25

## Contents

## 1 Revision History

| **Version** | **Description** | **Date** |
|---|---|---|
| 3 | Language specification done, added Implementation steps | 2025-03-05 |
| 2 | Most of the syntax done, draft of BNF | 2025-02-26 |
| 1 | Core idea and some syntax | 2025-02-19 |

# 2 Language Specification

## 2.1 Syntax and Code Style

```
# Example program written in FunkCode
# Single-line comments use "#"

##
Multiline comments use double "#".
The first occurrence of "##" marks the end of the block.
##

funk add = (numb x, numb y) { return x + y };
funk sub = (numb x, numb y) { return x - y };

funk main = {
    mut numb res = add(1, 2);
    res = sub(res, 1);
};

main();
```

## 2.2 Data Types

Funk is statically typed, requiring explicit type declarations for all variables. The auto keyword may be introduced for type inference in the future.

### 2.2.1 Primitive Types

- `numb`: Integer values.
- `real`: Floating-point values (uses . as the decimal separator).
- `bool`: Boolean values (true, false).
- `char`: Single-character values (enclosed in single quotes, e.g., ‘c’).
- `none`: Represents an absence of value.

### 2.2.2 Composite Types

- `text`: A sequence of char values (strings). Implemented natively or in Funk.
- `list`: A collection of values, using angle brackets to specify type (list).

### 2.2.3 Variables & Immutability

All variables are immutable by default. To allow mutation, use the `mut` keyword:

```
mut numb counter = 0;
counter = counter + 1;
```

Funk enforces ownership transfer in function return values. This means reassignment requires creating a new variable:

```
list<numb> numbs = [1, 2, 3];
list<numb> new_numbs = numbs.push(4);
```

## 2.3 Data Structures

Funk provides data structures as an alternative to traditional class-based inheritance. Data structures combine related data with associated functionality.

- Data structures are defined using the `data` keyword.
- Data structure variables cannot be reassigned after initialization.
- Fields are immutable by default unless declared with `mut`.
- All fields are public and directly accessible using dot notation (`structure.field_name`).
- Methods can access all fields of their containing data structure.
- All functions can read fields in data while only mutable functions can change mutable values.

Examples:

```
# Definition
data Rectangle {
    numb width;
    numb height;
    mut numb area = 0;

    # Method that reads fields but doesn't modify anything
    funk calculate_area = {
        return width * height;
    };

    # Method that updates a mutable field
    mut funk update_area = {
        area = width * height;
    };
};

# Initialization
Rectangle rect = Rectangle {
    width = 10,
    height = 20
};

# Usage
numb w = rect.width;
numb a = rect.calculate_area();
rect.update_area();
```

## 2.4 Generics

Funk supports generic type parameters, allowing developers to create reusable data structures and functions that work with multiple types. Generic types are declared using angle brackets with type parameter names:

```
struct Pair<A, B> {
    A first;
    B second;
};

funk swap<A, B> = (Pair<A, B> pair) {
    return Pair<B, A> {
        first = pair.second,
        second = pair.first
    };
};
```

## 2.5 Control Structures

### 2.5.1 Conditional Statements

Funk supports if, else if, and else for branching logic:

```
if (x < y) {
    "x is less than y" >> print;
} else if (x == y) {
    "x equals y" >> print;
} else {
    "x is greater than y" >> print;
}
```

### 2.5.2 Pattern Matching

Pattern matching with match simplifies multiple conditional checks. It is restricted to primitive types:

```
match (x) {
    case (1) { "One" >> print; }
    case (2) { "Two" >> print; }
    case (3)
    case (4)
    case (5) { "Three, Four, or Five" >> print; }
    none     { "Unknown value" >> print; }
}
```

### 2.5.3 Loops

Funk provides only `while` loops. Collections may implement `.each()` for iteration:

```
mut numb i = 0;
while (i < 100) {
    i += 1;
}
```

## 2.6 Functions

Functions in Funk can be assigned to variables or used as anonymous expressions. They are defined in two parts, arguments and code block, If no arguments are given, functions can be used without it.

### 2.6.1 Function Declaration

```
funk add = (numb a, numb b) { return a + b };
```

### 2.6.2 Mutable Functions

Regular functions are pure and cannot modify variables outside their scope. Mutable functions (`mut funk`) can:

- Modify `mut` variables from enclosing scope.
- Have side effects.
- Modify data structure properties passed to them.

### 2.6.3 Function Execution

```
numb result = add(10, 5);
numb nested = add(10, add(20, 30));
```

### 2.6.4 Anonymous Functions

```
add(1, { return rand() * 100 });
```

### 2.6.5 Pattern Matching in Functions

Pattern matching can be used for declarative function definitions:

```
funk factorial = (0) { return 1 };
funk factorial = (numb n) { return n * factorial(n - 1) };
```

### 2.6.6 Implicit Returns (Lambda Syntax)

Might be implemented later.

```
funk double = (x) -> x * 2;
```

### 2.6.7 Function Overloading

Overloading allows functions to be defined with different type signatures:

```
funk print = (numb x) { ... };
funk print = (text x) { ... };
```

## 2.7 Piping & Function Chaining

Funk supports function chaining via the `>>` operator. This allows the output of one function to become the input of the next:

```
numb total = add(10, 2) >> add(5) >> (numb sum) { return sum * 10 };
list<text> sorted_results = fetch_data() >> filter() >> sort();
```

For readability, chains can be split across multiple lines:

```
fetch_data()
    >> filter()
    >> sort()
    >> (list<text> lst) { lst.each((text line) { line >> print; }) };
```

## 2.8 Scope

### 2.8.1 Hierarchy

Funk implements scope in a hierarchical structure:

- Global scope: Contains all top-level declarations (functions, data structures, variables).
- Function scope: Each function creates a new scope containing its parameters and variables.
- Block scope: Created by control structures (if, while) and code blocks `{}` as a new scope.

### 2.8.2 Access

Variables are visible in their declaring scope and all nested scopes. Inner scopes can declare variables with the same name as outer scopes, overloading the outer variable in this and subsequent scopes.

### 2.8.3 Mutability

**Pure functions (default)**

- Can access and modify their local variables if also mutable.
- Can access but not modify their function parameters. Alternatively parameters are sent as copies and not references in which case they can modify them.
- Can access but not modify variables from enclosing scopes.
- Cannot have side effects.
- Can call other pure functions, but not mutable ones.

**Mutable functions (`mut`)**

- Have the same properties as Pure functions.
- Can modify `mut` variables from enclosing scopes and passed parameters.
- Can have side effects.
- Can call both pure and mutable functions.

## 3 BNF

| Symbol | Explanation |
|---|---|
| `<...> \| <...>` | OR |
| `<...>*` | One or more |
| `[ ... ]` | Optional |
| `R'...'` | Regex |

```
<program>            ::= <statement>*

<statement>          ::= <declaration>
                       | <expression> ';'
                       | <control>
                       | <comment>

<comment>            ::= '#' <wildcard>* <newline> | '##' <wildcard>* '##'

<declaration>        ::= <variable_decl> | <function_decl> | <data_decl>
<variable_decl>      ::= ['mut'] <type> <identifier> '=' <expression> ';'
<function_decl>      ::= ['mut'] 'funk' <identifier> '=' <function_def> ';'
<data_decl>          ::= 'data' <identifier> [<generics>] <data_block> ';'

<function_def>       ::= [ '(' [<param_list>] ')' ] <block> | '(' <literal> ')' <block>
<param_list>         ::= <param> [',' <param>]*
<param>              ::= <type> <identifier>
<block>              ::= '{' <statement>* '}'

<generics>           ::= '<' <identifier> [',' <identifier>]* '>'
<data_block>         ::= '{' [<data_member>]* '}'
<data_member>        ::= ['mut'] <type> <identifier> ['=' <expression>] ';'
                       | 'funk' <identifier> '=' <function_def> ';'

<control>            ::= <if_stmt> | <while_stmt> | <match_stmt> | <return_stmt>

<if_stmt>            ::= 'if' '(' <expression> ')' <block> ['else' <else_part>]
<else_part>          ::= <if_stmt> | <block>

<while_stmt>         ::= 'while' '(' <expression> ')' <block>

<match_stmt>         ::= 'match' '(' <expression> ')' '{' <case_list> '}'
<case_list>          ::= <case>*
<case>               ::= 'case' '(' <expression> ')' <block> | 'none' <block>

<return_stmt>        ::= 'return' <expression> ';'

<expression>         ::= <assignment> ['>>' <pipe_target>]*
<pipe_target>        ::= <identifier> | <call> | <anonymous_function>
<anonymous_function> ::= '(' [<param_list>] ')' <block>

<assignment>         ::= <logical_or> [<assignment_op> <expression>]
<assignment_op>      ::= '=' | '+=' | '-=' | '*=' | '/='
```

```
<logical_or>          ::= <logical_and> ['||' <logical_and>]*
<logical_and>         ::= <equality> ['&&' <equality>]*
<equality>            ::= <comparison> [('==' | '!=') <comparison>]*
<comparison>          ::= <additive> [('<' | '>' | '<=' | '>=') <additive>]*
<additive>            ::= <multiplicative> [('+' | '-') <multiplicative>]*
<multiplicative>      ::= <unary> [('*' | '/' | '%') <unary>]*
<unary>               ::= ('!' | '-') <unary> | <factor>

<factor>              ::= <literal>
                        | <identifier>
                        | <call>
                        | <list_literal>
                        | '(' <expression> ')'

<call>                ::= <identifier> '(' [<argument_list>] ')'
<argument_list>       ::= <expression> [',' <expression>]*

<list_literal>        ::= '[' [<expression> [',' <expression>]*] ']'

<literal>             ::= <number> | <boolean> | <char> | 'none' | <text>
<number>              ::= <integer> | <real>
<integer>             ::= <digit>*
<real>                ::= <digit>* '.' <digit>*
<boolean>             ::= 'TRUE' | 'FALSE'
<char>                ::= '\'' <wildcard> '\''
<text>                ::= '"' <wildcard>* '"'

<type>                ::= 'numb'
                        | 'real'
                        | 'bool'
                        | 'char'
                        | 'none'
                        | 'text'
                        | 'list' '<' <type> '>'
                        | <identifier>
                        | <identifier> '<' <type> [ ',' <type> ]* '>'

<identifier>          ::= <letter> [ <letter> | <digit> | '_' ]*
<wildcard>            ::= <digit> | <letter> | <other>

<newline>             ::= '\n'
<digit>               ::= R'0-9'
<letter>              ::= R'a-zA-Z'
<other>               ::= R'.'
```

# 4 Implementation

## 4.1 Overview

The project will be completed in 5 phases, which will be carried out sequentially to ensure sufficient time for essential and mandatory tasks.

1. Project setup in C++, including Tokens, Lexer, and basic Parser.
2. Implementation of all features required for a grade 3.
3. Implementation of more advanced features, still within the scope of the course, for a grade 5.
4. Language documentation and preparation for the presentation.
5. Addition of fun, interesting, and personal features to the project, if time permits.

## 4.2 Detailed Steps

1. Define Token list (Enum) from specification.
2. Implement Token Class with metadata from source code.
3. Lexer Implementation that converts source code to list of tokens.
4. Implement base node for all other nodes to inherit from.
5. Implement nodes for common datatypes.
6. Implement nodes for arithmetic operations.
7. Parser Implementation for evaluating nodes (beginning with arithmetic nodes).
8. Implement a basic print function for easier debugging. Will be correctly implemented later.
9. Extend AST with boolean nodes for boolean logic.
10. Implement scope manager (before variable implementation to prevent later rewrites).
11. Implement variable declaration and assignment.
12. Implement functions (rewrite print function?).
13. Handle parentheses and operator precedence.
14. Error handling and printing metadata such as col, row.
15. Control structures such as if-else and loops
16. Implement match function.
17. Implement mutations.
18. Implement data structures.
19. Implement function piping.
20. Implement pattern-matching for functions.
21. STL, type inference, ownership, unit testing, integration with IDE and more advanced features will be implemented last. Order not yet determined.

## 4.3 Distribution / Role Assignment

Most of the work will be done individually, but with frequent communication and key decisions made together. At the beginning of the project, we will work closely together to set up the lexer and parser, aiming to quickly create a minimal viable product before transitioning to more individual tasks.

The specific tasks each of us will take on have not yet been determined, but they will be assigned as the project progresses, with a focus on working on features that align with our individual interests.

## 4.4 Deadlines

| **Date** | **Task** |
|---|---|
| 2025-03-28 | Lexer complete, can convert source code to list of Tokens. |
| 2025-04-01 | Primitive datatypes implemented as AST Nodes. |
| 2025-04-04 | Basic arithmetic operations can be performed and evaluated. |
| 2025-04-09 | Basic boolen operations and comparisons completed. |
| 2025-04-11 | Assignment of variables in global scope completed. |
| 2025-04-16 | Function declaration and calling complete, with correct scope. |
| 2025-04-18 | Operator precedence complete, if not implemented before. |
| 2025-04-21 | Mutation implemented. |
| 2025-04-25 | If/Else + Loops implemented. |
| 2025-04-30 | Recursion implemented. Verify that project meets grade 3 criteria. |
| 2025-05-02 | Match + case implemented. |
| 2025-05-07 | Data structures implemented. |
| 2025-05-09 | Function piping implemented. |
| 2025-05-12 | Pattern matching implemented. (Optional, in case of extra time.) |
| 2025-05-12 | Documentation + Project complete. |
| 2025-05-13 | *Project deadline for presentation.* |
| 2025-06-02 | *Project deadline for document submission.* |