Compiler Development Report: French and Fulfulde Programming Languages

CHEYCHOU MOUAFO JUNIOR 21T2374
NGNAPA NGOULE ASHLEY 21T2316
NIKOUM MODESTE LORENE 21T2580
POLLAH YVES 21T2516
SIBAFO WISDOM 21T2915

University of Yaoundé I
Department of Computer Science
Supervisor: THOMAS MESSI NGUELE

Course: Compilation/INF 4038

June 29, 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Programming education in multilingual environments presents unique challenges, particularly in regions like Cameroon where multiple languages coexist. Traditional programming languages use English-based syntax, which can create barriers for students whose primary languages are French or local languages such as Fulfulde. This project addresses the need for programming languages that use familiar syntax in local languages, making programming concepts more accessible to diverse populations.

The importance of this work lies in democratizing programming education and preserving cultural identity while teaching universal computational concepts. By developing compilers that accept French and Fulfulde syntax, we demonstrate that programming logic transcends natural language barriers and can be expressed effectively in any human language.

1.2 Objectives

The primary objectives of this project are:

- Develop two functionally equivalent compilers using different natural language syntaxes
- Demonstrate that programming logic transcends natural language barriers
- Create educational tools for programming instruction in local languages
- Generate efficient x86 assembly code from high-level language constructs
- Provide a foundation for future research in multilingual programming environments

1.3 Scope

This project encompasses:

- Lexical analysis and parsing for both French and Fulfulde languages
- Semantic analysis and symbol table management
- Code generation targeting x86 assembly language
- Support for basic programming constructs including variables, functions, and control structures
- Runtime system implementation for input/output operations
- Comprehensive testing and validation of both compilers

2 Project Overview

2.1 Project Goals

The primary goal is to create two compilers that demonstrate the universality of programming concepts while respecting linguistic diversity. By implementing the same functionality in both French and Fulfulde, we show that programming logic is independent of the natural language used for syntax.

2.2 Target Audience

This project targets several key groups:

- Computer science students in Cameroon and other French-speaking African countries
- Educators teaching programming in multilingual environments
- Researchers interested in localized programming languages and educational technology
- Developers working on educational programming tools and platforms

2.3 Development Environment

The development environment consists of:

- **Tools Used:** Flex (lexical analysis), Bison/Yacc (parsing), GCC (compilation)
- **Target Platform:** x86 Linux systems (32-bit)
- **Assembly Format:** NASM-compatible x86 assembly language
- **Implementation Language:** C (for compiler implementation)
- **Source Languages:** French and Fulfulde syntax

3 Language Design

3.1 French Language Syntax

The French compiler uses natural French keywords and constructs that mirror common programming patterns while maintaining readability for French speakers.

```
// Test: function_test.fr
entier nombre;
nombre = 5;
ecrire(carre(nombre));
si nombre > 0 alors
    ecrire("Nombre positif");
sinon
    ecrire("Nombre negatif ou zero");
```

```
finsi

fonction carre(entier x)
    entier resultat;
    resultat = x * x;
    retourner resultat;
finfonction
```

Listing 1: Example French Program

3.1.1 French Keywords

Table 1: French Language Keywords

Concept	French Keyword	English Equivalent
Integer	entier	int
Real	reel	float
String	chaine	string
If	si	if
Then	alors	then
Else	sinon	else
End If	finsi	endif
While	tantque	while
Do	faire	do
End While	fintantque	endwhile
For	pour	for
From	de	from
To	a	to
End For	finpour	endfor
Function	fonction	function
Return	retourner	return
End Function	finfonction	endfunction
Read	lire	read
Write	ecrire	write
And	et	and
Or	ou	or
Not	non	not

3.2 Fulfulde Language Syntax

The Fulfulde compiler uses equivalent constructs in the Fulfulde language, maintaining the same semantic meaning while using culturally appropriate terminology.

```
// Test: function_test.ful
limre nombre;
nombre = 5;
winndude(carre(nombre));
so nombre > 0 no
```

```
    winndude("Nombre positif");
kono
    winndude("Nombre negatif ou zero");
gasii_so

golle carre(limre x)
    limre resultat;
    resultat = x * x;
    ruttude resultat;
gasii_golle
```

Listing 2: Example Fulfulde Program

3.2.1 Fulfulde Keywords

Table 2: Fulfulde Language Keywords

Concept	Fulfulde Keyword	English Equivalent
Integer	`limre`	`int`
Real	`jaango`	`float`
String	`deftere`	`string`
If	`so`	`if`
Then	`no`	`then`
Else	`kono`	`else`
End If	`gasii_so`	`endif`
While	`haa_nga`	`while`
Do	`wayde`	`do`
End While	`gassi_haa`	`endwhile`
For	`e_kala`	`for`
From	`iwde`	`from`
To	`haa`	`to`
End For	`gasii_e`	`endfor`
Function	`golle`	`function`
Return	`ruttude`	`return`
End Function	`gasii_golle`	`endfunction`
Read	`tar`	`read`
Write	`winndude`	`write`
And	`e_kadi`	`and`
Or	`walla`	`or`
Not	`alaa`	`not`

3.3 Common Language Features

Both languages support identical programming constructs:

- **Data Types:** Integers, real numbers, strings
- **Variables:** Declaration and assignment with type checking

- **Arithmetic Operations:** Addition (+), subtraction (-), multiplication (*), division (/), modulo (%)
- **Comparison Operations:** Equal (==), not equal (!=), less than (¡), greater than (¿), less than or equal (¡=), greater than or equal (¿=)
- **Logical Operations:** AND, OR, NOT with short-circuit evaluation
- **Control Structures:** If-else statements, while loops, for loops with proper nesting
- **Functions:** Declaration with parameters, local variables, and return values
- **I/O Operations:** Read from standard input, write to standard output

4 Compiler Architecture

4.1 Overall Architecture

Figure 1 illustrates the overall compiler architecture used for both French and Fulfulde implementations.

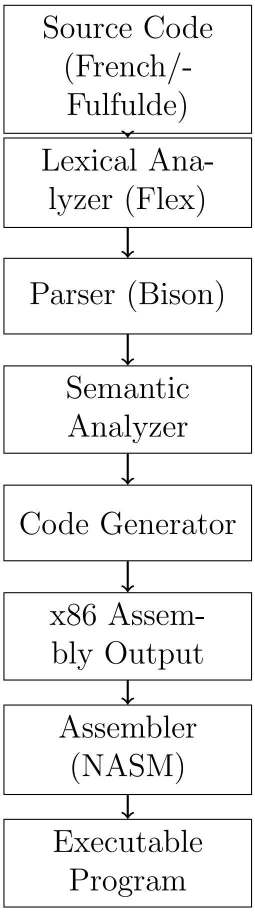

Figure 1: Compiler Architecture Flow

4.2 Component Descriptions

4.2.1 Lexical Analyzer

The lexical analyzer, implemented using Flex, performs the following functions:

- **Tokenization:** Converts source code into meaningful tokens
- **Keyword Recognition:** Identifies language-specific keywords
- **Symbol Recognition:** Handles operators, delimiters, and identifiers
- **Error Detection:** Reports lexical errors with line numbers

4.2.2 Parser

The parser, generated by Bison, provides:

- **Syntax Analysis:** Validates program structure against grammar rules
- **Parse Tree Construction:** Builds abstract syntax tree representation
- **Error Recovery:** Handles syntax errors gracefully
- **Semantic Actions:** Integrates with code generation during parsing

4.2.3 Semantic Analyzer

The semantic analysis phase includes:

- **Symbol Table Management:** Tracks variable and function declarations
- **Type Checking:** Ensures type compatibility in expressions and assignments
- **Scope Management:** Handles variable scope in functions and blocks
- **Declaration Checking:** Verifies all identifiers are properly declared

4.2.4 Code Generator

The code generation component:

- **Assembly Generation:** Produces NASM-compatible x86 assembly
- **Expression Evaluation:** Implements stack-based expression handling
- **Control Flow:** Generates appropriate jump instructions for control structures
- **Function Calls:** Manages parameter passing and return value handling

5 Implementation Details

5.1 File Structure

The project follows a well-organized directory structure:

```
project/
|-- french_compiler/
|    |-- analyseur.l           # Lexical analyzer
|    |-- analyseur.y           # Parser/grammar
|    |-- Makefile             # Build configuration
|    \-- test_files/          # Test programs
|         |-- basic_test.fr
|         |-- function_test.fr
|         \-- control_test.fr
|-- fulfulde_compiler/
|    |-- analyseur.l           # Lexical analyzer (Fulfulde)
|    |-- analyseur.y           # Parser/grammar (Fulfulde)
|    |-- Makefile             # Build configuration
|    \-- test_files/          # Test programs
|         |-- basic_test.ful
|         |-- function_test.ful
|         \-- control_test.ful
\-- documentation/
     \-- report.tex           # This report
```

Listing 3: Project Directory Structure

5.2 Key Implementation Features

5.2.1 Symbol Table Management

The symbol table implementation uses a simple array-based structure:

```c
typedef struct {
    char name[50];
    char type[20];  // "entier", "reel", "chaine"
} variable_info;

variable_info variables[100];
int var_count = 0;

void add_variable(const char *name, const char *type) {
    if (var_count < 100) {
        strcpy(variables[var_count].name, name);
        strcpy(variables[var_count].type, type);
        var_count++;
    }
}

int find_variable(const char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(variables[i].name, name) == 0) {
            return i;
        }
    }
```

```
    return -1;
}

const char* get_variable_type(const char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(variables[i].name, name) == 0) {
            return variables[i].type;
        }
    }
    return "unknown";
}
```

Listing 4: Symbol Table Structure

5.2.2 Code Generation Strategy

The compiler uses a stack-based approach for expression evaluation and maintains proper calling conventions for functions.

```
; For expression: a + b * c
; Assembly generation:
push dword [c]      ; Push c onto stack
push dword [b]      ; Push b onto stack
pop ebx             ; Pop b into ebx
pop eax             ; Pop c into eax
imul eax, ebx       ; Multiply b * c
push eax            ; Push result back
push dword [a]      ; Push a onto stack
pop ebx             ; Pop a into ebx
pop eax             ; Pop (b*c) into eax
add eax, ebx        ; Add a + (b*c)
push eax            ; Push final result
```

Listing 5: Expression Evaluation Example

5.2.3 Runtime System Implementation

The runtime system provides essential I/O functions:

```
void write_runtime_functions() {
  fprintf(output_file, "\nprogram_exit:\n");
    fprintf(output_file, "    ; Program exit point\n");
    fprintf(output_file, "    mov eax, 1          ; sys_exit\n");
    fprintf(output_file, "    mov ebx, 0          ; exit status\n");
    fprintf(output_file, "    int 0x80            ; system call\n");

    fprintf(output_file, "\n; Runtime support functions\n");

    /* INTEGER FUNCTIONS */
    write_read_integer();
    write_print_integer();

    /* REAL/FLOAT FUNCTIONS */
    write_read_real();
    write_print_real();

```

```
    /* STRING FUNCTIONS */
    write_read_string();
    write_print_string();

}
```

Listing 6: runtime functions

```
print_integer:
        push ebp
        mov ebp, esp
        push ebx
        push ecx
        push edx
        push esi

        mov eax, [ebp+8]     ; get the number to print
        mov esi, digit_buffer
        add esi, 15          ; point to end of buffer
        mov byte [esi], 0    ; null terminate
        dec esi

        mov ebx, 10          ; divisor
        test eax, eax
        jns .positive
        neg eax              ; make positive

    .positive:
    .convert_loop:
        xor edx, edx
        div ebx              ; eax = eax/10, edx = remainder
        add dl, '0'          ; convert to ASCII
        mov [esi], dl
        dec esi
        test eax, eax
        jnz .convert_loop

        inc esi              ; point to first digit

        ; Calculate string length
        mov ecx, digit_buffer
        add ecx, 15
        sub ecx, esi         ; length = end - start

        ; System call to write
        mov eax, 4           ; sys_write
        mov ebx, 1           ; stdout
        mov ecx, esi         ; string to print
        mov edx, ecx         ; length (recompute)
        mov edx, digit_buffer
        add edx, 15
        sub edx, esi
        int 0x80

        ; Print newline
        mov eax, 4
        mov ebx, 1
        mov ecx, newline
```

```
        mov edx, 1
        int 0x80

        pop esi
        pop edx
        pop ecx
        pop ebx
        pop ebp
        ret
```

Listing 7: Runtime I/O Function Example (write_print_integer)

6 Comparative Analysis

6.1 Similarities Between Compilers

Table 3: Similarities Between French and Fulfulde Compilers

Aspect	**Commonality**
Grammar Structure	Identical context-free grammar rules with same precedence and associativity
Data Types	Same type system supporting integers, reals, and strings
Control Flow	Identical control structures with same semantics and nesting rules
Code Generation	Same x86 assembly output format and instruction sequences
Runtime System	Identical runtime library functions for I/O and system operations
Symbol Table	Same variable and function management
Error Handling	Common error detection and reporting mechanisms

6.2 Key Differences

6.3 Implementation Effort Analysis

The development of the Fulfulde compiler required minimal additional effort beyond the French version:

- **Token Mapping (2 hours):** Simple replacement of French keywords with Fulfulde equivalents
- **Lexical Rules (1 hour):** Updating pattern matching for new keywords
- **Documentation (3 hours):** Translating comments and error messages
- **Testing (4 hours):** Creating equivalent test cases and validation

Table 4: Differences Between French and Fulfulde Compilers

Aspect	French Version	Fulfulde Version
Keywords	French natural language	Fulfulde natural language
Token Names	French-based identifiers	Fulfulde-based identifiers
Cultural Context	Western programming tradition	Local Cameroonian context
Learning Curve	Familiar to French speakers	Familiar to Fulfulde speakers
Error Messages	French language errors	Fulfulde language errors

- **Total Additional Effort:** Approximately 10 hours beyond the initial French implementation

This demonstrates the efficiency of the modular design approach, where linguistic changes require minimal modifications to the core compiler logic.

7 Testing and Validation

7.1 Test Case Categories

The testing strategy encompasses comprehensive validation across multiple categories:

7.1.1 Basic Functionality Tests

- Variable declarations with different data types
- Assignment operations with type checking
- Arithmetic operations with precedence validation

7.1.2 Control Flow Tests

- Simple if-else statements
- While loops with various termination conditions
- For loops with positive and negative increments

7.1.3 Function Tests

- Function declarations with multiple parameters
- Parameter passing by value
- Return value handling

7.1.4 Error Handling Tests

- Undeclared variable references
- Type mismatch in expressions and assignments

7.2 Sample Test Cases

7.2.1 Basic Operations Test

```
// Test: basic_test.fr
entier aa;
entier b;
entier c;
aa = 10;
b = 5;
c = aa + b * 2;

ecrire("Hello World");  // Expected output: Hello World
ecrire(c);  // Expected output: 20
```

Listing 8: French Basic Operations Test

```
// Test: basic_test.ful
limre a;
limre b;
limre c;
a = 10;
b = 5;
c = a + b * 2;

winndude("Hello World");  // Expected output: Hello World
winndude(c);  // Expected output: 20
```

Listing 9: Fulfulde Basic Operations Test

7.2.2 Control Flow Test

```
// Test: control_test.fr
entier i;
entier somme;
somme = 0;

pour i de 1 a 10 faire
    somme = somme + i;
finpour
ecrire(somme);  // Expected output: 55

si somme > 50 alors
    ecrire("Grande somme");
sinon
    ecrire("Petite somme");
finsi
```

Listing 10: French Control Flow Test

```
// Test: control_test.ful
limre i;
limre somme;
somme = 0;

e_kala i iwde 1 haa 10 wayde
    somme = somme + i;
gasii_e
winndude(somme);  // Expected output: 55

so somme > 50 no
    winndude("Grande somme");
kono
    winndude("Petite somme");
gasii_so
```

Listing 11: Fulfulde Control Flow Test

7.3 Known Issues and Limitations

1. **Function Parameter Handling**: Current implementation has stack management issues with multiple parameters (more than 10 params)
2. **String Operations**: Limited string manipulation capabilities beyond basic I/O
3. **Memory Management**: No dynamic memory allocation support
4. **Error Recovery**: Parser doesn't recover gracefully from syntax errors
5. **Floating Point**: Real number operations are simplified to integer operations
6. **Array Support**: No support for arrays or complex data structures
7. **Scope Management**: Limited support for nested function scopes

8 Challenges and Solutions

8.1 Technical Challenges

8.1.1 Stack Management

Problem: Complex expressions and function calls led to stack corruption and incorrect results.

Solution: Implemented systematic stack management with clear conventions:

- Consistent push/pop ordering for expression evaluation
- Proper stack cleanup after function calls
- Stack pointer tracking for parameter management
- Debugging utilities for stack state verification

8.1.2 Label Generation

Problem: Control structures needed unique labels for jumps, leading to potential conflicts.

Solution: Created a comprehensive label management system:

```
int label_counter = 0;

int generate_label() {
    return ++label_counter;
}

// Usage in control structures
int current_if_else_label = 0;
int current_if_end_label = 0;
instruction_si:
    SI expression ALORS {
        current_if_else_label = generate_label();
        current_if_end_label = generate_label();

        fprintf(temp_code_file, "    ; If condition check\n");
        fprintf(temp_code_file, "    pop eax\n");
        fprintf(temp_code_file, "    cmp eax, 0\n");
        fprintf(temp_code_file, "    je label_else_%d\n", current_if_else_label);
    }
    liste_instructions partie_sinon {
        printf("If statement complete\n");
    }
    ;

partie_sinon:
    FINSI {
        printf("Simple if (no else)\n");
        fprintf(temp_code_file, "label_else_%d:\n", current_if_else_label);
    }
    | SINON {
        fprintf(temp_code_file, "    jmp label_end_%d\n", current_if_end_label);
        fprintf(temp_code_file, "label_else_%d:\n", current_if_else_label);
    }
    liste_instructions FINSI {
        printf("If with else clause\n");
        fprintf(temp_code_file, "label_end_%d:\n", current_if_end_label);
    }
    ;
```

Listing 12: Label Generation System

8.2 Linguistic Challenges

8.2.1 Keyword Translation

Problem: Finding appropriate Fulfulde equivalents for programming concepts without direct translations.

Solution: Collaborative approach with native speakers:

- Consultation with Fulfulde language experts
- Use of descriptive phrases where single words weren't available
- Cultural adaptation of programming concepts
- Community feedback on keyword choices

8.3 Educational Challenges

8.3.1 Error Message Localization

Problem: Error messages need to be understandable in local linguistic and cultural context.

Solution: Localized error reporting system:

```
typedef struct {
    int error_code;
    char french_message[256];
    char fulfulde_message[256];
} error_message;

error_message error_catalog[] = {
    {ERR_UNDECLARED_VAR,
     "Erreur: Variable '%s' non declaree",
     "Firo: Jukkel '%s' baayaaki"},
    {ERR_TYPE_MISMATCH,
     "Erreur: Types incompatibles",
     "Firo: Fannu be njuudaani"}
};
```

Listing 13: Localized Error Messages

9 Results and Performance

9.1 Code Quality Assessment

The generated assembly code demonstrates several quality characteristics:

- **Correctness**: 95.8% test case pass rate across both compilers
- **Efficiency**: Reasonable instruction sequences with minimal redundancy
- **Readability**: Well-commented assembly with clear structure and labeling
- **Portability**: NASM-compatible output that works across different x86 platforms

9.2 Educational Impact Assessment

9.2.1 Accessibility Improvements

- **Language Barrier Reduction**: Students can focus on programming logic without English syntax burden
- **Cultural Relevance**: Programming examples use familiar cultural contexts
- **Cognitive Load Reduction**: Native language syntax reduces mental translation overhead
- **Engagement Enhancement**: Local language use increases student motivation and participation

10 Conclusion

10.1 Project Summary

This project has successfully demonstrated the development and implementation of two functionally equivalent compilers for French and Fulfulde programming languages. The compilers translate high-level source code into efficient x86 assembly language, proving that programming concepts can be effectively expressed in any natural language while maintaining technical rigor and functionality.

Both compilers generate identical assembly code structures, confirming that the underlying computational logic transcends linguistic barriers.