engineering proceedings

MDPI

Proceeding Paper

Combining Forth and Rust: A Robust and Efficient Approach for Low-Level System Programming [†]

Priya Gupta [1], Ravi Rahar [2], Rahul Kumar Yadav [2], Ajit Singh [2], Ramandeep [2] and Sunil Kumar [3,*]

[1] Atal Bihari Vajpayee School of Management and Entrepreneurship, Jawaharlal Nehru University, New Delhi 110067, India; priyagupta@jnu.ac.in
[2] School of Engineering, Jawaharlal Nehru University, New Delhi 110067, India; ravi56_soe@jnu.ac.in (R.R.); rahul99_soe@jnu.ac.in (R.K.Y.); ajit99_soe@jnu.ac.in (A.S.); ramand53_soe@jnu.ac.in (R.)
[3] Department of Computer Science & Information Technology, Central University of Haryana, Mahendragarh 123031, India
* Correspondence: drsunilk@cuh.ac.in

† Presented at the International Conference on Recent Advances in Science and Engineering, Dubai, United Arab Emirates, 4–5 October 2023.

Abstract: Rust is a modern programming language that addresses the drawbacks of earlier languages by providing features such as memory safety at compilation and high performance. Rust's memory safety features include ownership and borrowing, which makes it an ideal choice for systems programming, where memory safety is critical. Forth is a stack-based programming language that is widely used for low-level system programming due to its simplicity and ease of use. This research paper aims to explore the combination of Forth and Rust programming languages to create a more robust and efficient solution for low-level system programming. The primary objective is to demonstrate the implementation of essential Forth operations, including addition, subtraction, assignment, comparison, and if-else statements, while demonstrating loops, push operations, and dump operations in Rust. The implementation of these operations in Rust is demonstrated using code from actual implementation. This research paper also discusses the advantages of using Rust for low-level system programming. Rust's memory safety features, coupled with its high performance, make it an ideal choice for systems programming, where memory safety and performance are critical. The combination of Forth and Rust provides a more efficient and safer solution for low-level system programming, making the implementation more robust. Our implementation tries to leverage these properties of both languages to make a memory-safe and low-level system programming language. This research paper also includes code snippets to provide a practical demonstration of how the Forth operations can be implemented in Rust.

Keywords: rust; forth; compiler; memory safe; stack-based

Citation: Gupta, P.; Rahar, R.; Yadav, R.K.; Singh, A.; Ramandeep; Kumar, S. Combining Forth and Rust: A Robust and Efficient Approach for Low-Level System Programming. *Eng. Proc.* **2023**, *59*, 54. https://doi.org/10.3390/engproc2023059054

Academic Editors: Nithesh Naik, Rajiv Selvam, Pavan Hiremath, Suhas Kowshik CS and Ritesh Ramakrishna Bhat

Published: 17 December 2023

1. Introduction

Forth is a stack-based programming language that was first developed in the late 1960s by Charles H. Moore. The language was originally designed to be a tool for rapid prototyping of computer systems, but it quickly gained popularity in low-level system programming due to its simplicity and efficiency. Forth evolved from Charles H. Moore's personal programming system, which had been in continuous development since 1968 [1]. Forth's minimalistic syntax and stack-based data structure make it easy to learn and understand, even for programmers with limited experience in low-level programming. Forth's data structure is based on a stack, where operands are pushed onto the stack and operators manipulate the operands on the stack [2]. This allows for a concise and elegant syntax that is easy to read and write. In addition, the stack-based architecture makes it easy to implement complex algorithms using a series of simple operations. Forth's small memory footprint and direct control over hardware make it suitable for embedded systems and real-time applications. Rust, on the other hand, is a modern programming language that was developed by Mozilla in 2010. Rust is designed to be safe, concurrent, and fast, with a strong emphasis on memory safety. Rust's unique ownership and borrowing system ensures that programs are memory-safe at compile time, eliminating many common programming errors [3].

Rust's memory safety features make it an ideal choice for low-level system programming, where memory errors can have serious consequences. Rust also provides powerful concurrency features that make it well-suited for high-performance, multithreaded applications [4]. By combining Forth and Rust, it can leverage the simplicity and elegance of Forth with the safety and efficiency of Rust. This research paper will explore the implementation of various Forth operations using the Rust programming language. The later part will discuss the implementation of basic arithmetic operators, comparison operators, conditional statements, loop constructs, and stack operations using Rust's built-in operators and control flow constructs.

Forth can offer the responsiveness necessary for critical tasks where real-time control and safety are paramount, while Rust can handle background processing and sophisticated algorithms. This combination makes sure that potential memory bugs or security flaws will not affect real-time operations. Rust can be used for computationally intensive parts in applications that require high performance, such as simulations, scientific computing, or data processing, while Forth can be integrated for low-level hardware interactions and control. This allows for effective memory usage and improved performance without compromising security. Integrating Rust can offer contemporary tooling, package management, and safety improvements in systems with pre-existing Forth codebases.

By checking array bounds at runtime and preventing access to out-of-bounds memory, Rust prevents buffer overflows. Buffer overflow attacks, which are frequently used to gain unauthorized access or run arbitrary code, must be stopped from doing this. Forth and Rust interfaces can be created to offer secure and controlled points of interaction. This can stop unsafe operations from being carried out or unauthorized data manipulation. Processes for code auditing and validation are made easier by Rust's strong type system and memory safety features. Knowing that Rust's compiler checks will have already obliterated some classes of vulnerabilities allows security audits to concentrate on high-risk areas. In this analysis of a combined solution leveraging Rust's ownership and borrowing model for memory management, the impact is evident despite being non-direct. Rust's stringent ownership system enhances memory safety, preventing common issues like double-free errors, while its borrowing mechanism and lifetimes ensure safe concurrent data access. In contrast, traditional Forth implementations, reliant on manual memory management, lack Rust's safety features, potentially leading to memory-related vulnerabilities and making the analysis an essential exploration of memory management paradigms.

The implementation of basic arithmetic operators such as addition, subtraction, multiplication, and division is straightforward in Forth. These operators manipulate the values on the stack, and the result is pushed back onto the stack [2]. The implementation in Rust is equally simple, using Rust's built-in arithmetic operators to perform the necessary calculations. Comparison operators such as less than, greater than, and equal to are also commonly used in Forth programming. These operators compare values on the stack and push a Boolean result back onto the stack. In Rust, comparison operators can be implemented using the standard comparison operators such as ==, !=, <, >, <=, and >=.

Conditional statements, such as if-else statements, are used to execute different code blocks based on a Boolean condition. In Forth, conditional statements can be implemented using the if-else construct, which tests a condition and executes the appropriate code block. Rust provides a similar if-else construct that can be used to implement conditional statements. Stack operations, such as push and pop, are essential to Forth programming. These operations manipulate the stack, pushing values onto the stack and popping values off the stack. Rust provides a similar stack manipulation functionality using its built-in vector data structure.

Background

The Forth programming language is a stack-oriented language that was created in the late 1960s. It is known for its simplicity and flexibility, which has made it a popular choice for embedded systems and real-time applications [5]. However, its syntax and semantics can be challenging for new users, and its lack of a formal specification has led to multiple implementations with different features and behaviors [6].

One popular implementation of Forth is porth, which is written in Python [7]. Porth is designed to be portable, extensible, and easy to read and modify. It supports most of the standard Forth primitives, as well as some Python-specific features such as generators and lambdas [7]. However, Python may not be the most efficient language for implementing Forth, especially for performance-critical applications [7]. Rust, on the other hand, is a systems programming language that provides a balance between safety, speed, and expressiveness [8]. Rust's ownership and borrowing system helps prevent common programming errors such as null pointers and data races, while its performance is comparable to that of C and C++.

In this context, the implementation of a Forth compiler in Rust inspired by porth can provide several benefits. First, it can leverage Rust's performance and safety features to create a fast and reliable Forth implementation that can be used in various domains, from microcontrollers to web applications. Second, it can extend porth's functionality by adding new primitives, libraries, and tools that are specific to Rust, such as support for multithreading, SIMD instructions, and foreign function interfaces. Third, it can provide a reference implementation of Forth in Rust that can help educate and inspire developers who are interested in systems programming and low-level optimization.

The code snippets provided in this study show some of the key features of the proposed Forth compiler. These features reflect the standard Forth semantics and syntax, while also incorporating some Rust-specific idioms (e.g., the use of Option types for optional references). Overall, the implementation of a Forth compiler in Rust can be a valuable contribution to the Forth and Rust communities, as well as a fascinating exploration of the intersection between two powerful and versatile languages.

Rust and Forth can be combined to optimize various functionalities and determine the system's performance-critical components that could use Rust's optimization. These might involve memory-intensive tasks, difficult algorithms, or data processing, using Forth for other tasks and integrating Rust components specifically for these parts. The ownership model in Rust can aid in effective memory management. For data structures that demand dynamic memory allocation, it is best to use Rust. These structures can be interfaced with Forth, which enables Rust to manage memory allocation and deallocation. Rust supports in-line assembly for precise hardware control. Utilizing this feature can seamlessly integrate Rust with Forth's direct hardware interaction by inserting low-level assembly code where required. With Forth's direct hardware interaction, everything runs smoothly. Implementing complex algorithms in Rust will take advantage of its performance optimizations. Forth can call these Rust functions when needed, allowing your system to achieve high efficiency without sacrificing Forth's hardware control. Comparative analysis of memory safety features can be seen in Table 1.

Table 1. Comparative Analysis of Memory Safety Features of Rust.

Memory Safety Feature	Description
Ownership and borrowing	The ownership mechanism in Rust makes sure that each memory variable has a distinct owner. Data races are avoided through borrowing restrictions, which also provide secure concurrent access by avoiding multiple mutable references to the same data.
Lifetimes	The lifetimes system in Rust upholds rules on the duration of a reference's validity, avoiding dangling references and memory-related issues. References are always kept current, and deallocated memory is not accessed.
Safe Concurrency	By avoiding data races with its ownership and borrowing restrictions, Rust upholds thread safety. Data exchange and transmission across threads is made safe via the Send and Sync characteristics.
Safe Memory Management	Rust's memory management system ensures strong bounds checking and forgoes uncontrolled access to protect against buffer overflows, stack overflows, and other memory-related problems.

This study will move on to the comparison operators, like greater than and equal to. It will discuss the implementation code and Rust code used to implement each operator, after which the discussion of if-else constructs and loops will be included. Again, it will demonstrate how Rust's built-in control flow constructs can be used to implement these constructs. Finally, it will explore the implementation of stack operations such as push, dump, and dup. It will discuss how Rust's built-in vector data structure can be used to implement these operations.

2. Materials and Methods/Methodology

To implement the various Forth operations using Rust, it has used Rust's built-in operators and control flow constructs. One of the most important features of Forth is its stack data structure. Rust's built-in vector data structure is an excellent tool to implement stack operations like push and pop. The vector data structure in Rust is efficient and provides automatic memory management, making it easy to use [7]. To implement the arithmetic operations, it has used Rust's built-in arithmetic operators, such as addition (Algorithm 1), subtraction, multiplication, and division. These operators have allowed us to perform basic arithmetic operations on the operands present on the stack. The equality and inequality operators can be used to implement the comparison operators. Rust provides multiple comparison operators like '==' (equal to), '!=' (not equal to), '>' (greater than), '<' (less than), '>=' (greater than or equal to), and '<=' (less than or equal to) [2]. For now, only the greater than operator has been implemented.

Algorithm 1. Implementation of plus (+) operator

```
 let a = handle_stack_empty(stack.pop(),token);
let b = handle_stack_empty(stack.pop(),token);
stack.push(a + b);
```

To implement conditional statements, it has used Rust's built-in if-else control flow construct (Algorithm 2). It can evaluate a condition based on the top of the stack and perform an action accordingly. This will allow us to implement conditional statements in Forth. In the implementation below, => indicates an arm of a match statement which matches the word of the next instruction.

Algorithm 2. Implementation of ‘if’, ‘else’, and ‘end’ block

```
Word::OpIf(else_end_idx) => {
    let a = handle_stack_empty(stack.pop(), token);
    if a == 0 {
        let Some(else_end_idx) = else_{end_idx
        else {println!(“Error: ‘if’ does not have
        reference to end of block”); exit(1)};
        token_idx = else_end_idx - 1;
    }
}
Word::OpElse(end_idx) => {
    let Some(end_idx) = end_idx
    else {println!(“Error: ‘else’ does not have
    reference to end of block”); exit(1)};
    token_idx = end_idx - 1;
}
Word::OpEnd(wile_end_idx) => {
    let Some(wile_end_idx) = wile_end_idx
    else {println!(“Error: ‘end’ does not have
    reference to while block or next instruction”);
    exit(1)};
    token_idx = wile_end_idx - 1;
}
```

To implement the loop constructs, it has used Rust’s built-in loop construct and conditional statements. It can use the loop construct to create an infinite loop and then use conditional statements to break out of the loop when a certain condition is met. Rust also provides a while loop construct that can be used to loop through a set of instructions while a condition is true.

Overall, the implementation of various Forth operations using Rust provides a unique and modern approach to low-level system programming. By leveraging the simplicity and elegance of Forth with the safety and efficiency of Rust, it can create reliable and high-performance low-level systems. Current implementation can run in two modes: simulation mode and compilation mode. In simulation mode, Rust’s operations and data structures are used to simulate operations like Forth. On the other hand, in compilation mode, the Forth program is converted to assembly and then compiled to binary, also generating object files in the process. The ELF64 format is used for binary generation. Simulation mode is implemented in the simulated_program() function and compilation mode is implemented in the compile_program() function. Both functions take a vector of struct type Token (Algorithm 3) which contains the filename in which it appeared, the line number(row), the place in line(col) where it appeared, and the actual lexeme which was present.

Algorithm 3. Structure of struct type Token

```
struct Token {
    file_path: String,
    col: usize,
    row: usize,
    word: Word,
}
```

The lexing is performed by the function lex_file(). It generates tokens from files. Tokens can contain any one of the variants of enum word (Algorithm 4), which implements all of the keywords that are allowed in the language [8]. The current implementation has twelve words. The words for ‘If’, ‘else’, ‘end’, and ‘do’ need to store the index of the next operation so that that can be jumped to if needed based on the condition provided. This is where the powerful enums of Rust shine. The same indexes are used in assembly in compilation mode to be saved as addresses so that they can be jumped to if needed.

Algorithm 4. Elements of enum word

```
enum Word {
    OpPush(i32),
    OpPlus,
    OpMinus,
    OpEqual,
    OpDump,
    OpDup,
    OpGt,
    OpIf(Option<usize>),
    OpEnd(Option<usize>),
    OpElse(Option<usize>),
    OpWhile,
    OpDo(Option<usize>),
}
```

The Forth language implementation uses postfix notation [2] that can be described in Algorithms 5–7 below:

Algorithm 5. Sample Forth program and its output

```
40 20 +
40 20 −
40 20 =
40 20 >

Output:
60
20
 0
 1
```

Algorithm 6. Sample Forth program and its output

```
1 1 = if 5 . end
1 0 = if 5 . end

Output:
5
```

Algorithm 7. Sample Forth program and its output

```
5 while dup 0 > do
dup .
1 -
end

Output:
5
4
3
2
1
```

First the lexing is performed and each word is broken and identified as a token and stored in the token struct [9]. All tokens thus generated are stored in a vector. Finally, blocks are checked to see if they are properly closed. 'If (else)' blocks and 'while (do)' blocks are closed by the end. If they are not closed properly, then an error is generated. In the case of the 'If' block, the vector indices of 'if', 'else', and 'end' are stored. In the case of the 'while' block, the indices of 'while', 'do', and 'end' are stored [10]. They are stored in the same word enum. For example: In the case of 'if . . . end' in simulation mode, the index of 'end' is stored in 'if'. If the after 'if' statement is true, the execution continues, and if it is false, the instruction next to 'end' is executed. This cross referencing is performed by another function crossreference_blocks(). The result of this function, which is still a vector of tokens, is passed to simulation mode or compilation mode as requested by the user [11].

It also looks at how to implement comparison operators like greater than '>' and equal to '=' in Rust. The pseudocode for the greater than, less than, and equal to operators involves popping two values from the stack, comparing them, and pushing the result back onto the stack [12]. This demonstrates how Rust's built-in comparison operators can be used to implement comparison operations in a straightforward and easy-to-understand way. It also implements loop constructs using Rust's built-in control flow constructs. The pseudocode for a while loop involves checking the length of the stack and executing a code block if the stack is not empty. This illustrates how Rust's built-in control flow constructs can be used to implement loop constructs in a clear and concise manner.

Finally, it explores how to implement stack operations using Rust's built-in vector data structure. The pseudocode for the pop operation involves popping a value off the stack. On the other hand, to implement compilation mode, nasm assembly is used. Its keywords are used. For example: jmp and jz are used for unconditional and conditional (when zero) jumps. Addresses are provided to all instructions as they are executed to make it easy to jump back to them [13].

3. Results

The implemented Forth compiler in Rust provides a solid foundation for future investigations and developments. Its successful completion highlights the feasibility and advantages of integrating Rust's modern programming capabilities with the functionality of Forth. This accomplishment not only reinforces Rust's reputation as a powerful language for systems programming but also encourages further exploration of how Rust can be leveraged in the implementation of other low-level programming languages or systems software. Because Forth and Rust have different design philosophies, toolchains, and runtime environments, combining them poses portability challenges. It takes careful planning to ensure broad applicability while controlling platform-specific dependencies. Here is how to handle these difficulties: Rust has a more robust runtime and standard library than Forth, which frequently uses a stack-based interpreter as its runtime environment [14]. Runtime environment compatibility can be difficult. Memory management models used by Forth and Rust are dissimilar. Determining clear data exchange formats and interfaces is necessary to ensure seamless memory exchange. The implementation of the Forth compiler using Rust demonstrates the efficacy of combining these two programming languages. The resulting compiler exhibits improved reliability, performance, and memory management, thanks to Rust's unique features and capabilities. This research paper sets the stage for future research endeavors in the realm of utilizing Rust for implementing low-level programming languages and systems software.

There are several advantages of using Rust with Forth. Rust is a memory-safe programming language that eliminates the possibility of null and dangling pointer errors [11]. This feature makes Rust a great language to use with Forth, as Forth does not have a garbage collector and relies heavily on memory management [15]. Rust is designed to be a fast language that provides low-level control over system resources. This feature makes it an ideal language to use with Forth, as Forth is a language that requires speed and efficiency. Rust's syntax is expressive and readable, making it easier for developers to understand and maintain code. This feature makes Rust a good choice to use with Forth, which is known for its concise and expressive syntax. Rust can be easily integrated with C and C++, which means that it can be used with existing Forth codebases that are written in these languages. This feature makes Rust a versatile language to use with Forth.

Despite several advantages, it also suffers from some disadvantages. Rust has a steeper learning curve compared to other programming languages, which can be a disadvantage for developers who are new to Rust and Forth. While Rust has a growing ecosystem of libraries and tools, it still has a smaller library collection compared to other programming languages like Python and Java. This can be a disadvantage for developers who require specific libraries for their Forth projects. Because Forth uses stacks and direct memory manipulation, it is challenging to measure memory utilization correctly [16]. Forth's low-level memory operations can collide with Rust's ownership and borrowing mechanism, which places a strong emphasis on security. Forth is frequently employed in embedded and real-time systems where exact timing is essential [17]. Rust's execution time profiling might provide overhead that hinders Forth's real-time capabilities. To provide useful findings, profilers frequently rely on Symbolic Information (e.g., function names). It is challenging to trace profiling findings back to the original Forth source code since Forth code is often interpreted or compiled into a low-level representation. Profilers can be resource-intensive, making them unsuitable for contexts where Forth is often used due to resource limitations.

4. Discussion

This research paper offers a comprehensive exploration of integrating Rust and Forth, shedding light on the advantages and challenges that arise from merging these two programming languages. In terms of prospects for this research, several possibilities are worth considering.

Firstly, there is a potential for further exploration of real-world use cases. While the paper already outlines some potential applications for integrating Rust and Forth, it is likely that numerous other use cases remain unexplored. To broaden our understanding, future research endeavors could examine how these languages can be effectively utilized in different contexts, such as the development of embedded systems, operating systems, or various software applications. Investigating performance gains represents another promising avenue. The combination of Rust and Forth offers the potential for improved performance. To delve deeper into this aspect, future research could conduct benchmarking tests to measure the speed and efficiency of code written using this integrated approach, in comparison to other programming languages or alternative methods. The integration process itself could benefit from refinement. While this research paper provides a starting point for integrating Rust and Forth, there may be room for improvement.

Future investigations could focus on making this integration more seamless by developing new tools, libraries, or frameworks that simplify the process and enhance its efficiency. Additionally, it would be valuable to compare this approach with others aiming to achieve similar goals. Although this research paper looks into the intricacies of integrating Rust and Forth, alternative approaches may exist. To determine the most effective strategies across different contexts, future research could undertake comparative analyses, contrasting various approaches to identify their respective strengths and weaknesses.

5. Conclusions

The task of this research paper was to investigate the implementation of a Forth compiler using Rust, a modern programming language whose cornerstones are its robustness and efficiency. The choice of Rust is due to its characteristics such as its unique ownership model and borrowing rules which endow it with suitable capabilities for implementing low-level programming languages such as Forth. By leveraging Rust's memory management and error handling capabilities, the Forth compiler implemented in this paper is reliable and less prone to bugs. This study involved designing a parser and lexer for Forth, creating a runtime environment, and integrating these components to build the compiler.

The resulting Forth compiler in Rust is a powerful system that can operate on multiple platforms. The successful implementation of this compiler highlights Rust's potential as a language for systems programming and opens possibilities for further research into how Rust can be utilized in implementing other low-level programming languages or systems software.

Author Contributions: Conceptualization, R.K.Y. and P.G.; methodology, R.R.; software, R.R.; validation, P.G. and S.K.; formal analysis, A.S.; investigation, A.S.; resources, R.; writing—original draft preparation, R.K.Y.; writing—review and editing, R.K.Y.; visualization, R.R. and R.K.Y.; supervision, P.G. and S.K.; project administration, R.K.Y. and P.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gibson, R.G.; Bergin, T.J. *History of Programming Languages-II*; ACM Press: New York, NY, USA, 1996.
2. Scanlon, L.J. *Forth Programming*; Howard W.Sams & Co.: Indianapolis, IN, USA, 1983.
3. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques & Tools*; Braille Jymico Inc.: Charlesbourg, QC, Canada, 2015.
4. Klabnik, S.; Nichols, C. *The Rust Programming Language*; No Starch Press: San Francisco, CA, USA, 2023.
5. *Encyclopedia of Computer Science*; Wiley: Chichester, UK, 2008.
6. Qin, B.; Chen, Y.; Yu, Z.; Song, L.; Zhang, Y. Understanding memory and thread safety practices and issues in real-world rust programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London, UK, 15–20 June 2020.
7. Available online: https://www.complang.tuwien.ac.at/anton/euroforth/ef14/papers/ertl.pdf (accessed on 3 November 2023).
8. Brodie, L. *Thinking Forth*; Punchy Pub.: London, UK, 2004.
9. Brodie, L.; FORTH Inc, C.O.R.P.O.R.A.T.E. *Starting Forth*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1987.
10. Brakefield, J.C. Challenges for Forth. In Proceedings of the Second and Third Annual Workshops on Forth—FORTH '90 and '91, San Antonio, TX, USA; 1991.
11. The Rust Programming Language. Available online: https://doc.rust-lang.org/book/ (accessed on 3 November 2023).
12. Available online: https://pages.physics.wisc.edu/~lmaurer/forth/Forth-79.pdf (accessed on 3 November 2023).
13. 83 Standard. Available online: https://forth.sourceforge.net/standard/fst83/ (accessed on 3 November 2023).
14. Thompson, C. How Rust Went from a Side Project to the World's Most-Loved Programming Language. Available online: https://www.technologyreview.com/2023/02/14/1067869 (accessed on 3 November 2023).
15. Olney, R.; Benson, M. *Forth Techniques*; Pan: London, UK, 1985.
16. Forsely, L.P. *1988 Rochester Forth Conference: Programming Environments, June 14–18, 1988, University of Rochester*; The Institute for Applied Forth Research: Rochester, NY, USA, 1988.
17. Katzan, H. *Invitation to Forth*; Petrocelli Books: Princeton, NJ, USA, 1981.