PROGRAMMING LANGUAGE IN C

“PIRI”

Pietro Rodrigano, Angel Lopez, Noah valderrama and Jose Miguel Serrano

1. Target audience and needs

Audience: Older adults, possibly with limited exposure to programming.
Needs: Easy-to-understand syntax, clear error messages, and a focus on simplicity.

2. Purpose of the Language

Educational: A tool for learning basic programming concepts.
Practical: Could be used for simple automation or data processing tasks.
Engagement: Engage older adults in cognitive exercises through programming.

3. Key Features

Simple Syntax: A syntax that is easy to read and write, avoiding complex constructs.
Limited Set of Features: Basic programming constructs like variables, simple data types (integers, strings), conditionals, loops, and basic input/output.
Error Handling: Clear and friendly error messages.

Now that we had all this we started creating the project.

1. Developing a Lexer (Tokenizer):

We first implemented a lexer that tokenizes the input code into meaningful components. Writing regular expressions to recognize keywords, identifiers, literals, and symbols.

In Lexer (lexer.c):

- Natural Language Keywords:

Keywords like create, show, if, then, repeat are used instead of traditional programming keywords. This makes the language more relatable and easier to understand for someone not familiar with conventional programming terminologies.

- Descriptive Variable Declarations:

Variables are declared with create number [variableName] set to [value], which is more descriptive and intuitive than using int or other traditional data type keywords.

- Comments:

Comments begin with --, a simple and clear way to denote comments, which is easy to remember and use.

- Error Tokens:

The lexer includes an error token type to handle any unrecognized or problematic inputs, ensuring robust error handling.

In Lexer Header (lexer.h):

- Clear Enumerations and Structure Declarations:

The header file declares token types and the token structure clearly, ensuring that the lexer's functionality is well-defined and accessible to other parts of the language implementation.

Function Declarations:

Functions for creating tokens and tokenizing source code are declared, providing a clear interface for the lexer's operations.

2. Building a Parser:

Then, we developed a parser that built an abstract syntax tree (AST) from the tokenized input.

- AST Structures: We needed to define the structures that would make up our AST. Each type of expression or statement in the language would have a corresponding structure.

- Parsing Functions: For each rule in our language's grammar, we would have a parsing function. These functions will look at the current tokens and decide how to parse them into AST nodes.

- Error Handling: The parser we created is able to handle and report syntax errors gracefully.

In Parser:

- Intuitive Syntax Parsing:

The parser is designed to interpret the tokens according to the language's natural language-like syntax, handling variable declarations, display statements, conditional statements, loops, and arithmetic operations in a way that mirrors everyday language.

- Descriptive Error Messages:

The parser includes mechanisms to provide clear and informative error messages, guiding users through correcting syntax errors.

- Simple Control Structures:

Conditional statements use if ... then ... structure, and loops use repeat ... times, which are more intuitive than traditional programming control structures.

- Focus on Readability and Simplicity:

The entire syntax and structure of the parser are geared towards readability and simplicity, avoiding complex programming constructs.

3. Implementing the Interpreter:

The interpreter for Piri is a crucial component that executes programs written in the language. It operates by traversing the Abstract Syntax Tree (AST) generated by the parser, executing each node according to Piri's syntax rules.

This interpreter handles various constructs such as variable declarations, conditional statements, loops, and output operations, translating them into executable actions. A significant feature of Piri's interpreter is its focus on error handling. It is designed to provide clear and instructive error messages, crucial for the target audience of older adults. These messages guide users through troubleshooting syntax or logical errors, making the programming experience more approachable and less intimidating for beginners.

4. Error Handling

The error handling mechanism in Piri is designed to provide robust and user-friendly error management. By incorporating `setjmp` and `longjmp`, Piri introduces an exception-like handling system, similar to try-catch blocks found in other languages.

When an error occurs, such as division by zero, the program does not crash; instead, it triggers a custom error handler that logs the error message and then redirects the flow of execution back to a safe state.

This approach ensures that errors are handled gracefully, maintaining the program's stability and providing clear feedback to the user, which is especially beneficial for older adults who are new to programming.

Conclusion

In conclusion, the Piri programming language, designed for older adults, incorporates a natural language-like syntax for ease of understanding. Despite implementing a lexer, parser, and interpreter, Piri didn't fully achieve its desired impact. The lexer effectively tokenizes code, and the parser builds an understandable AST. The interpreter executes this AST. However, the language faced challenges in completely aligning with the needs of its target audience, highlighting the complexities in creating a programming language that balances simplicity, functionality, and user engagement.