Natural Language Programming Language - Technical Manual

Jack Farell 20352136 Conall Kavanagh 20478414

0. Table of contents

1. Introduction

1.1 Overview

For our third-year project, we created an interpretive programming language with that is easy to learn for non-programmers to create their programs. As well as creating and testing the programming language to ensure it is functional we also tested the programming language with people who are not familiar with programming to see if our language succeeds in allowing them a greater understanding of code and the logic that is needed for future languages.

The goals of our project are to

- Create a programming language
- Create an interpreter for the language
- Test the language to see if it is easier to learn when compared to other languages e.g. python, java, javascript, C

After six weeks of the project, we have completed the goals for the project and are happy with the outcome of the project which will be revealed throughout the document.

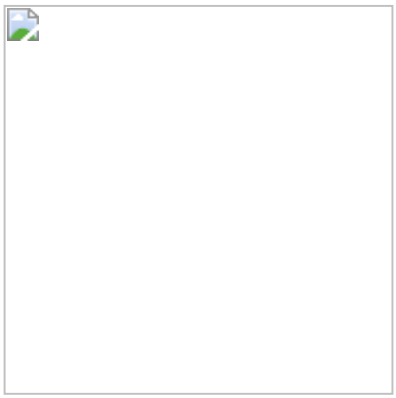
1.2 Motivation The motivation for us deciding to do this was that we had an interest in programming and how programming languages work so we wanted to experience programming languages from a different point of view. The reason for our choice to make the language easy to learn was that we wanted to pick a specific demographic that would use a language but a language for them didn't exist already. Choosing a specific demographic for the language it also influenced how the language was made and presented to people, It also changed our way of thinking since we have more experience with programming and the syntax required it is second nature so we had to think back to when we started coding and what would have helped us back then.

1.3 Glossary

- **Lexer** - the process of taking a sequence of characters and turning them into tokens with assigned meanings.
- **Parser** - Taking the tokens created by the lexer and constructs the abstract syntax tree.
- **Abstract Syntax Tree (AST)** - A tree representation of the source code in normal language, that can be modified to change the final implementation of the source code.
- **Interpreter** - a program that executes instructions given to it by source code without it having to be compiled into a lower-level language.
- **Compiler** - a program that takes a source code file and translates it to a lower-level language for the computer to understand.

2. System Architecture

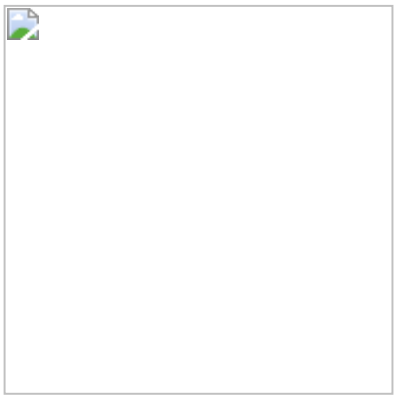
This section describes the high-level overview of the system architecture showing the distribution functions across (potential) system modules. Architectural components that are reused or 3rd party should be highlighted. Unlike the architecture in the Functional Specification - this description must reflect the design components of the system as it is demonstrated.



The coder will start by creating their source code file using the language, the language will then be turned from its plain text state into a series of tokens by the lexer which will then be used by the parser to create the abstract syntax tree (AST). an AST is a tree representation of the source file that is used to create and manipulate the code. The AST provides us with enough information about the structure of the code that we can choose what outcome we want based on how we manipulate the code. Now that we have the code made we can now run the code and using an interpreter the code will be brought down to a lower language for the computer to understand the request and execute them.

3. High-Level Design

The coder will start by creating their source code file using the language, the language will then be turned from its plain text state into a series of tokens by the lexer which will then be used by the parser to create the abstract syntax tree (AST). an AST is a tree representation of the source file that is used to create and manipulate the code. The AST provides us with enough information about the structure of the code that we can choose what outcome we want based on how we manipulate the code. Now that we have the code made we can now run the code and using an interpreter the code will be brought down to a lower language for the computer to understand the request and execute them.



From the diagram above we can see that when the user inputs their code file into the language it first goes through the lexer which breaks up the text file into tokens that can then be read by the parser in conjunction with the grammar file to form a parse tree which shows the relationship with each of the tokens created by the lexer. The parse tree is then read by the visitor who finds each of the tokens and then runs the corresponding code that was written in the visitor file.

4 functional requirements

Requirement	**Variables**
Description	Variables that can store values of any type described by the language
Criticality	High
Technical issues	Need to validate that the values are of a valid type, and needs to support expressions, which will get evaluated at runtime, should not be callable outside of their scope
Dependencies with other requirements	Data types supported by language, functions, which should have a separate scope.

Requirement	**Loops**
Description	Loops that can repeat things for a certain number of steps
Criticality	High
Technical issues	Need to work with either a number literal or a variable
Dependencies with other requirements	N/A

Requirement	**Procedure Definition**
Description	Define procedures in the code for code reuse
Criticality	medium
Technical issues	Procedures need their own scope and have to be passed parameters which are copied to the local scope. They should end on a return statement and return its body, or if there isnt one, on the end of the function body they should return null

Requirement	Procedure Definition
Dependencies with other requirements	Scoping, return statement

Requirement	Procedure Invocation
Description	Procedure that have been previously defined shall be able to be invoked
Criticality	medium
Technical issues	Must return the value in the procedure to the outer scope and output must be able to be stored
Dependencies with other requirements	Procedure definition, return statement, scoping

Requirement	If statements
Description	If statements are control flow which run a block if an expression evaluates to true
Criticality	High
Technical issues	If statement can have any number of `else if` statement and may optionally be followed by an `else`. all further else blocks mustn't be evaluated if one is found true and we need to separate the blocks in a way that we can unambigously determine which if statement they are a part of. It needs to be unambigous which If statement an `else` is a part of (dangling else problem)
Dependencies with other requirements	Booleans, expressions

Requirement	Expressions
Description	Expressions are combinations of terms and operators that will evaluate to a token in one of the language's datatypes
Criticality	High - needed for any mathematical logic
Technical issues	Need to ensure that all terms in an expression are of compatible datatypes. Need to ensure the correct order of operations
Dependencies with other requirements	Booleans, Strings, Numbers, Lists

Requirement	Strings
Description	Strings are a combination of characters used to represent text
Criticality	High
Technical issues	Need to ensure that they are not used in expressions with other incompatible data types
Dependencies with other requirements	Expressions, Say command

Requirement	Numbers
Description	Numbers in our programming language. All stored as floats internally so users dont have to distinguish between ints and floats
Criticality	High
Technical issues	Need to ensure that they are not used in expressions with other incompatible data types. When printed if they are a whole number, we shouldn't print `.0`

Requirement	Numbers
Dependencies with other requirements	Expressions

Requirement	Booleans
Description	True or False values
Criticality	High - needed for boolean logic
Technical issues	Need to ensure the correct order of operations for boolean expressions
Dependencies with other requirements	Expressions

Requirement	Scoping
Description	Variables in separate scopes (procedure bodies) should not be able to be accessed outside of the scope
Criticality	medium
Technical issues	Requires a stack in order to separate different scopes in memory. when we go into a new scope we need to push to a stack and when we go out of a scope we need to pop from the stack
Dependencies with other requirements	Function definition, function invocation

Requirement	Return statements https://scratch.mit.edu/ (https://scratch.mit.edu/)
Description	Return a value and exit the procedure body
Criticality	medium
Technical issues	need to differentiate between return statements and other statements in function bodies so we know to exit the function body and return the value
Dependencies with other requirements	function definition, function invocation

Requirement	Print statements
Description	print to the screen a string or variable given to it
Criticality	high
Technical issues	Need to make a string representation of all datatypes
Dependencies with other requirements	all datatypes

Requirement	Operators
Description	common mathematical, boolean and comparison operators
Criticality	High
Technical issues	need to ensure order of operations, must check that terms are of compatible data types. operators may work differently depending on the types
Dependencies with other requirements	Expressions, data types

Requirement	Installation
Description	Installing the interpreter
Criticality	High
Technical issues	Requires installing dependencies (Java, ANTLR)
Dependencies with other requirements	Java v17.x, ANTLR v4.X

5. Problems and Resolution

5.1 Setting up the project environment

A problem that we first encountered at the very beginning of the project was setting up antlr4 which was needed to create the language. The problems arose from having to set up the windows subsystem for Linux which required Jack to go into the motherboard settings to enable virtualisation. Then there were issues with file permissions trying to download the antlr4 jar file but after a bit of research and learning of the `!!` command that ignores file permissions the environment for the project was set up.

5.2 Learning antlr4

Other problems that arose were simply trying to figure out how antlr4 works, with neither of us having used antlr4 before we were simultaneously learning how to use antlr4 while learning how to make a language. This caused any problem with the language to become bigger as we had to figure out if it was a mistake we made in java with the logic of the command or if it was a problem with how we structured the grammar file as it was through a mistake and a meeting with our supervisor that we were told that antlr4 reads its grammar file from the top down and things closer to the top take precedence over things underneath.

5.3 Debugging with dynamic typing

We also had problems with some of our visitors. We found that while dynamic typing made it easier to get an interpreter up and running in the first place, it made it harder to debug things later on as we had to ensure that each variable was of a compatible type at runtime. We solved this by having a checker in the visitor file that saw what kind of variable it was and then depending on that would determine what to do with it.

I.e if two variables are added together it checks to see if it should do mathematical addition or if it needs to do string concatenation.

5.4 Scope

We had problems at first in creating functions and ensuring that they were in a separate scope from other parts of the program. We resolved this by implementing the memory as a Stack. When we go into a new scope, we will push and new HashMap onto the stack. When we leave the scope, we pop off the stack. This ensures that procedures have their own scope and do not affect variables outside of their scope.

5.5 if else blocks For our if statements, due to the way we were originally parsing them, when it came time to implement them in our interpreter we had access to the list of statements but no way of determining which ‘block’ it was a part of. This required us to go back to the parser, and instead of having if statements be parsed like this:

```
ifstmt: IF expression do stm* end (ELSE IF expression do stm* end)* (ELSE do stm* end)? ;
```

This made all the statements end up in the same list, making it not possible to determine which block they were from so we changed it to this:

```
ifstmt: IF expression block (ELSE IF expression block)* (ELSE block)? ;

block: DO stm* END;
```

This separated the statements into individual blocks in the AST and allowed us to separate them and know unambiguously which block they were a part of.

6. Installation Guide

As the language uses antlr4 which can be run on windows machines it should be possible for the language to function, however, we did not make the language with windows machines in mind as we have been using WSL for language creation and testing, It is assumed that you will be installing this on a Linux machine or have WSL installed.

1. Install WSL (https://learn.microsoft.com/en-us/windows/wsl/install (https://learn.microsoft.com/en-us/windows/wsl/install)) (For windows users)
2. Install Java (version 17.X or higher): https://www.java.com/en/download/help/download_options.html (https://www.java.com/en/download/help/download_options.html)

> *For Linux users this should be very simple as all that is need is* `sudo apt install default-jdk` *3. Install ANTLR4 using this guide: https://github.com/antlr/antlr4/blob/master/doc/getting-started.md#unix (https://github.com/antlr/antlr4/blob/master/doc/getting-started.md#unix) This is a simple guide that only requires you to copy the commands and paste them either into the command line or into the startup file if you plan on using the language regularly 4. Clone the repository at: https://gitlab.computing.dcu.ie/farrej82/2023-ca326-jfarrell-ckavanagh/ (https://gitlab.computing.dcu.ie/farrej82/2023-ca326-jfarrell-ckavanagh/)*
>
> - *When you click on the link you will be taken to the git lab page for the language,*
> - *Click on fork button which can be found at the top right of the page*
> - *That will make a copy of the repo in your own personal GitLab so make sure you are on that version before continuing on.*
> - *Click on the blue “Clone” button and a drop-down menu will appear.*
> - *Click on whichever option you prefer, for this example we will use HTTPS as it doesn't require any setup to be done beforehand.*
> - *Next, go to the terminal on your Linux machine and type in* `git clone *copy the link here*.`
> - *This should make the repository folder appear in the current directory you are in. 5. In your terminal go to the repository root folder. 6. type* `cd src/code` *into the command line. This takes you to where you can run the code. 7. type* `./run.sh` *into the command line. This generates and compiles the files needed for the language to work. 8. You can now run the interpreter by typing* `java placeholder path/to/file. 9. Try` *java placeholder test/hello_world` for a simple hello-world example.*

7 Testing

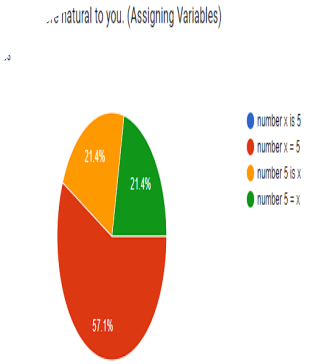
We did continuous testing of our parser and interpreter with the test examples in `src/code/test`. We tested the parser using ANTLR and running the parser in gui mode using our run.sh script like so:

```
./run.sh prog test/filename -gui
```

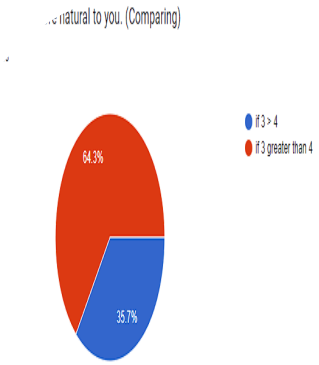
We did user testing using a survey, where we gave them multiple examples of possible syntax and asked them to pick which one felt more natural to them. We then applied this feedback to our grammar so that we could fit our syntax with what felt most natural to them.

In this survey, we found some interesting things about the syntax that they preferred.

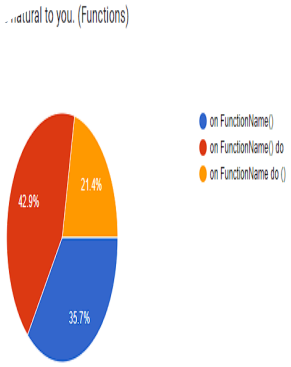
For the assignment of variables. Our users in general preferred the syntax `number x = 5` with 57.1% of them choosing this. Nobody chose `number x is 5` which was the one that we thought made the most sense at first. And the rest of them liked `number 5 is x` and `number 5 = x` equally, with the same percentage choosing both.



For comparison operators, Our users did not like the common mathematical operators for these expressions and much preferred them to be written in words. 64.3% of them preferred `if 3 greater than 4` over `if 3 > 4`. 78.6% of them preferred `3 not equal to 4` for what is the `!=` operator in most programming languages.



Users also preferred using `on functionname () do` as the way of declaring functions. So we decided to change the grammar of our language to use `do … end` syntax instead of curly braces `{}` to represent blocks of code.



We then did a second, final user test where we gave them some simple beginner programming tasks to carry out. After we did these exercises we gave them a few questions to fill out to evaluate if they felt that the language was easy to understand and learn. From the results of that survey, most of the users found our language easy learn. All of them agreed it was easy to understand and all of them strongly agreed that they would use this language if they were learning to code.