Alexander Smith

[Company name] | [Company address]

[Document title]

[Document subtitle]

# Abstract

As the field of computer science expands to include ever more methodologies and specialities, from mathematical proofs to artificial intelligence, weather modelling to financial modelling, general purpose programming languages end up being the Swiss army knife of tools, ok at everything, but poor at any one specific job. One such speciality is data processing, covering everything from initial extract, transform and load workloads, to big data exploration and back to tabulation of complex results.

Further, general purpose languages offer many solutions to data processing, often encouraging the user to read an entire file into memory and processing it many times into new memory structures. This methodology if often good enough for “simple” tasks, but rarely scales well. On the other hand, efficient streaming or more modern generator paradigms can be harder to implement when the programmer is not familiar with them.

“Hoshie Lang” is a new programming language that specialises in processing data. It has the following goals:

1. Enable the programmer to succinctly define what they want to happen in their data process.
2. Provide the programmer with a built-in library of common algorithms and functions used to process and analyse data.
3. Provide a pleasant programming experience in an IDE.
4. Encapsulate the execution code generation in a way that can be extended and adapted in the future to ensure Hoshie Lang can be executed on a variety of platforms.
5. Remove the need for the programmer to be familiar with memory efficiency and streaming / generator techniques.

## Succinct Programming

The Hoshie Lang programming language is based around a “Pipeline” analogy. The programmer is encouraged to think of the data as “flowing” through a pipe (one row at a time), which in turn is made up of:

* **Activities**: A section in the pipe that alters the data in some way, typically:
  + Altering the content of a row.
  + Augmenting or Truncating the format of a row.
  + Removing specific rows.
  + Appending new rows.
* **Sensors**: A section in the pipe that observes the data as it passes that point in the pipeline, these are typically **reducers** which calculate scalar values based on the data it has seen up to that point in time.

The language is **declarative**, which places a natural emphasis on breaking down the entire process into a smaller set of independent **actions**, which in turn can then be used in specific data processing pipelines.

Similarly, **sensors** can be defined and placed along the pipeline to observe the state of process at any given time.

Rectangle = Actions  
Ellipses = Sensors

## Built-in Activities and Sensors

The language includes a set of predefined functions that are used to define the actions and sensors. For example, some common actions are:

* **Filter**: Allow a subset of rows to flow through the pipe.
* **Map**: Maps a specific row structure to a new row structure.
* **FirstN**: Only allow N rows to pass through.

And some common Sensors would be:

* **Count**: A simple row count.
* **Max**: The max value of a nominated field.
* **Mean**: The mean value of a nominated field.

## IDE

For languages to become popular and be usable, they need to have good integrated development environment support as this lowers the barrier of entry when learning the language and can increase the programmer’s productivity by an order of magnitude.

## Encapsulated Code Generation

One of the visions for Hoshie Lang is to be able to process data on a variety of platforms. For example:

* Small data could be processed within a web browser.
* Medium data could be processed within a middleware application, NodeJS, Python or Java.
* Large data could be processed in a distributed computing cluster like Hadoop or Spark.

By ensuring the code generation is well encapsulated adding new code generators can be a relatively easy exercise, preferably by domain experts in those target platforms. For this paper, the reference implementation is targeting JavaScript which can cover the small to medium data sizes above.

## Efficient Runtime

Having an efficient runtime will be an important part of Hoshie Lang. In the reference JavaScript implementation, a streaming based runtime is used, making heavy use of JavaScript generators. This effectively means that data rows are “pulled” from the end of the pipe, triggering the upstream actions to do “just enough” work for each row to be created.

For example, if the source dataset is 1 billion rows in length, but the processing pipeline contains a “FirstN” action (e.g., one which only lets the first 100 rows through), then the bulk of source dataset will never be processed.

## Conclusion

There are a lot of moving parts when creating a new programming language, all of which play a key part to the overall usability of the language.

There are also a lot of tools, libraries, and IDEs available to assist in the task. This paper will take an in depth look at these and how they can be utilised to create a final product. Specifically:

* Language design.
* Compiler compilers, lexers and parsers.
* IDE integration and APIs
* Data processing and streaming techniques
* Documentation, unit test and build tools

# Declaration

# Acknowledgements

Table of Contents

[Abstract 1](#_Toc67743466)

[Succinct Programming 2](#_Toc67743467)

[Built-in Activities and Sensors 2](#_Toc67743468)

[IDE 2](#_Toc67743469)

[Encapsulated Code Generation 3](#_Toc67743470)

[Efficient Runtime 3](#_Toc67743471)

[Conclusion 3](#_Toc67743472)

[Declaration 4](#_Toc67743473)

[Acknowledgements 4](#_Toc67743474)

[Introduction 7](#_Toc67743475)

[Analysis 9](#_Toc67743476)

[Syntax Design **Error! Bookmark not defined.**](#_Toc67743477)

[Compiler **Error! Bookmark not defined.**](#_Toc67743478)

[Integrated Development Environment **Error! Bookmark not defined.**](#_Toc67743479)

[Runtime Environment **Error! Bookmark not defined.**](#_Toc67743480)

[Design 12](#_Toc67743481)

[Syntax 13](#_Toc67743482)

[Compiler 14](#_Toc67743483)

[Design Conclusion 14](#_Toc67743484)

[Implementation 15](#_Toc67743485)

[Development Environment 15](#_Toc67743486)

[Hoshi Lang Parsing 16](#_Toc67743487)

[Hoshie Lang Grammar 17](#_Toc67743488)

[Creating the Abstract Syntax Tree 21](#_Toc67743489)

[Navigating the Abstract Syntax Tree 21](#_Toc67743490)

[Integrated Development Environment 21](#_Toc67743491)

[Contribution Points 21](#_Toc67743492)

[Evaluation 22](#_Toc67743493)

[Conclusions 22](#_Toc67743494)

[Appendix 22](#_Toc67743495)

[References 22](#_Toc67743496)

[References 23](#_Toc67743497)

# Introduction

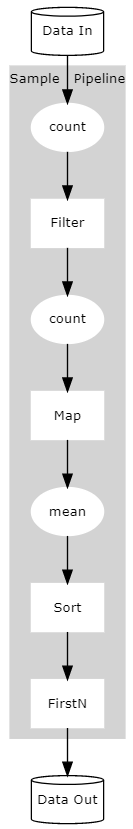
The overall goal of “Hoshie Lang” was to create a new programming language that focused on the task of Data Processing.

Having spent time working on several class assignments, intern projects and hackathons there was a common need to load one or more raw data files and perform some initial processing and analytics to help understand, interrogate and finally format the data before we could attend to the actual job or assignment at hand. Invariably this step was implemented in a very ad hoc way as it was a secondary concern, which ended up with the developer rewriting similar code in different circumstances and often with different programming languages.

Further, while programming languages usually provide plenty of support to get the job done, the programmer will often take the easiest approach which may be fine for smaller datasets but will fail to scale with the data efficiently. For example, they may load the entire dataset into memory and use a FOR loop to iterate over it, when a streamed reader / writer would be more efficient but more complex to code.

Finally, with any quick ad hoc code comes the additional risk of introducing subtle bugs, especially when the task appears to be secondary to the programmer.

Hoshie Lang attempts to solve these issues by presenting a data processing centric programming language. By design it encourages the programmer to define **what** data processing and analytics to perform on the data, without having to specify the **how** to do it. For example, Hoshie Lang has no keywords to “LOOP” or “STREAM” data, instead it has an abstraction of a “Pipeline”, one where data flows from the top of the pipe to the bottom, while passing through a series of “Activities” and “Sensors”, where an “Activity” alters the data in some way and a “Sensor” observes the data as it passes.



Figure

An example of a Simple Data Pipeline

This conceptually shows how the data flows from top to bottom, through a data processing “Pipeline”.

Each “Activity” is represented by a rectangle.

Each “Sensor” is represented by an ellipse

**Activity**: Typically changes the format of data (MAP) or the content of the dataset (FILTER).

**Sensor**: Observes the data as it passes that point in the “Pipeline”, typically reducing the data to a single value or set of values (COUNT, MEAN).

Of course, having a nice abstraction for data processing is only the first part of the puzzle. The next consideration is the target platform that the new language can execute on. As a student programmer the primary use case was with relatively small datasets on client machines, but it is worth considering other target platforms like SQL for traditional databases, Python for Machine Learning and AI activities or even Hadoop or Apache Spark for big data processing.

Finally, with any new language the programmer will expect a nice programming environment or integrated development environment, with syntax checking and highlighting features.

In summary this project creates a new programming language, with a nice development experience generating code that can be executed in Web Browsers and Node JS environments.

# Analysis

To get a clear idea of the project objectives, I discussed the idea with several peers, online friends and software professionals. While these conversations were mostly casual in nature, they did cover lots of interesting perspectives and really helped flesh out a better overall picture of what would be considered a Minimal Viable Product (MVP), or in this case a set of achievable objectives for this project.

The first step was to identify the **target audience**, while student computer scientists was top of the list, it became clear there was potentially a wider group of interested parties:

* Software Developers
* Data Scientists
* Database Administrators
* Computer System Operations
* Office Workers

These same targets could be further categorised into general competencies:

* Command Line Users
* Scripting
* Programming
* Querying (SQL)
* Integrated Development Environment Users
* Application users
* Linux
* Windows
* OSX

After further discussions, the following (biased and approximate) competency table was created:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Programmers** | **Data Sci.** | **DB Admin** | **Ops** | **Office** | **Total** |
| **CLI** | 8 | 6 | 7 | 10 | 1 | 32 |
| **Scripting** | 8 | 6 | 7 | 10 | 4 | 35 |
| **Programming** | 10 | 8 | 6 | 6 | 1 | 31 |
| **Query** | 7 | 9 | 10 | 4 | 4 | 34 |
| **IDE** | 10 | 9 | 8 | 7 | 1 | 35 |
| **Apps** | 5 | 7 | 5 | 4 | 8 | 29 |
| **Linux** | 7 | 6 | 7 | 10 | 1 | 31 |
| **Windows** | 7 | 7 | 6 | 6 | 6 | 32 |
| **OSX** | 7 | 7 | 6 | 4 | 6 | 30 |
| **Total** | 69 | 65 | 62 | 61 | 32 |  |

Summarising this table, the following points became clearer:

* Expertise – Should appeal to Programmers, Database Query Designers and Command Line / Scripting folks.
* Integrated Development Environment support would be needed.
* Operating System – A cross platform solution would be preferable.

To appeal to as wide an audience as possible support for a cross platform IDE was going to be necessary, the following short list was taken from the 2019 Stack Overflow Survey (Most Popular Development Environments, 2019):

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Popularity** | **Extensible** | **Cross Platform** |
| **Visual Studio Code** | 50.7% | Y | Y |
| **Visual Studio** | 31.5% | N | N |
| **Notepad++** | 30.5% | Y | N |
| **IntelliJ** | 25.4% | N | Y |
| **Vim** | 25.4% | Y | Y |
| **Sublime Text** | 23.4% | Y | Y |
| **Android Studio** | 16.9% | N | Y |
| **Eclipse** | 14.4% | Y | Y |
| **PyCharm** | 13.4% | N | Y |
| **Atom** | 13.3% | Y | Y |

At this point Visual Studio Code was picked as the target IDE, it also helped that I personally have used Visual Studio Code as my preferred development environment over the previous years.

The next major decision was how to parse Hoshi Lang. The first choice was to either roll our own custom code, or to use a tool to assist. In general this parsing is commonly referred to as a “**Compiler Compiler**” and there are several well established tools:

|  |  |
| --- | --- |
|  | **Target Language** |
| **Lex / Yacc** | C |
| **Flex / Bison** | C |
| **Antlr** | C++, C#, Dart, Go, Java, JavaScript, PHP, Python, Swift |

After some further research online, it became clear that Antlr was the modern popular choice currently and given its variety of target languages it became the “Compiler Compiler” of choice.

What language to develop in? From my previous studies and work experience, I felt comfortable developing this project in either **Python** **JavaScript** or **Java**. Antlr could target all these languages so that was fine, but to develop an extension for Visual Studio Code one would need to code in JavaScript or (preferably) TypeScript (TypeScript is an extension to JavaScript, adding “types” to make it more robust, it compiles to regular JavaScript so maintains compatibility with Web Browsers and NodeJS). After further research I satisfied myself that learning TypeScript would be relatively straight forward (having previous experience with JavaScript) as well as being beneficial (its popularity is rapidly growing). So, the decision was made to use **TypeScript** as the primary development language.

At this point picking a suitable **Target Language** for Hoshie Lang to compile to was an obvious choice, **JavaScript** was picked primarily to help keep the project manageable, but also because it was a good choice for small to medium sized data sets. It was noted however that it would be desirable to potentially target other languages in the future and therefore any Code Generation code should be completely encapsulated.

# Design

The mile high goal is relatively simple:

* Open IDE
* Create / edit a Hoshie Lang file (.ho)
* Press button to Compile and Run

Breaking it down further:

* Design Hoshi Lang Syntax
* Specify Hoshi Lang Syntax in ANTLR:
  + Lexer
  + Parser
* Transform output of ANTLR to something useful (Abstract Syntax Tree -> Concrete Syntax Tree)
* Code Generator (Concrete Syntax Tree -> JavaScript)
* Visual Studio Code Hoshie Lang Extension

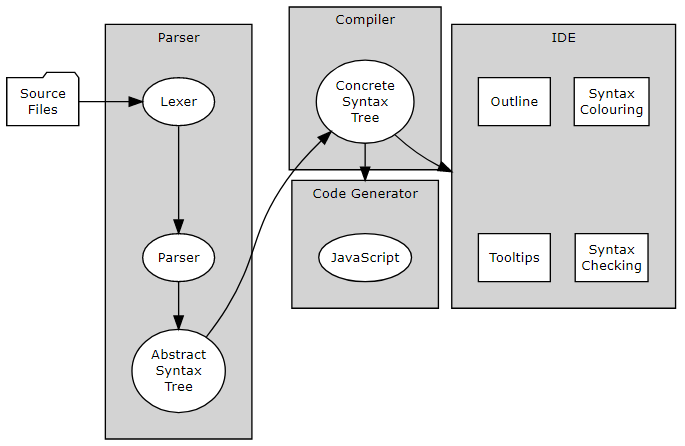


Figure - Overview

## Hoshi Lang Syntax

The design of the Hoshi Lang Syntax was done in two phases:

1. General:
   1. Type of language: Procedural, Object Orientated, Declarative etc.
   2. Scoping Rules: Visibility of identifiers etc.
2. Specific: The actual syntax of the language.

### Type of Language

From very early on the decision to make Hoshie Lang a **declarative** / **functional** language seemed an obvious one, since the goal was to create a language that focused on specifying what needed to be done and not how to do it, having a language which declared statements of truth in a simple expressive way, without the need of complex structures like OOP or even LOOPS seemed appropriate. When describing this to new users the following code would be articulated as “z is equivalent to x plus y”:

z = x + y;

In a procedural language the above statement would be “z equals x plus y at the time it is executed”.

The next high level decision was to make the language **immutable**, that is once an identifier has bee declared, it cannot be changed. Since the programmer doesn’t explicitly control the flow of the program, this is a natural design choice for a declarative language.

Finally all function declarations are **pure**, this mandates that the function should:

1. Always return the same result when the input parameters are the same.
2. Have no external side effects when calling the function.

### Scoping

Scoping rules are an important consideration when designing a language. The following rules were decided on:

Two distinct scopes:

1. File Level Scope:
   1. All declarations within a file are visible to all declarations declared after them.
   2. Declarations can be explicitly exported from a file.
   3. Exported declarations from one file can be explicitly imported
2. Function Level Scope:
   1. Passed parameters are visible within the body.
   2. Declarations within the function block are visible to items declared after them within the function block.

The key areas for the language are:

* Types:
  + Primitive: boolean, number, string
  + Complex: A named “set” of Primitives / Complex items
  + Arrays: A “list” of Complex or Primitive items (limited to one type)
* Expressions: A single entity that can be evaluated to a value:
  + Boolean: true, false, equality, greater, less, and, or, not etc.
  + Numeric: Plus, minus, multiplication, division, etc.
  + String: Concatenation
* Functions: An encapsulated set of expressions:
  + Gated inputs (parameters)
  + Internals should be private.
  + Single return value.
  + Should be pure.
* Declaration: A named Type, Expression or Function
* Scoping:
  + Files: Declarations inside a file will only be visible within that file unless a declaration is specifically exported.
  + Functions:
    - Declarations within the function are only visible within that function.
    - Declaration outside the function will not be visible within that function.
    - Parameters will be used to pass in expressions / declarations.
    - Return value will be used to expose a result from the function.
    - Functions are expected to be pure.
* Built-in Functions: The language will have a set of data processing functions built into the language. It will also have built-in support for unit testing.

\*\*MORE\*\*

## Compiler

\*\*MORE\*\*

* Grammar / Lexer
* AST -> CST

\*\*MORE\*\*

## Code Generator

\*\*MORE\*\*

## Visual Studio Code Hoshie Lang Extension

\*\*MORE\*\*

\*\*MORE\*\*

* Lexer: The set of known tokens, keywords, and operators
* Grammar: Defines how the tokens can be structured in a high-level way

\*\*MORE\*\*

## Design Conclusion

\*\*MORE\*\*

Primary development language would be TypeScript and the targeted runtime environment would NodeJS / Browser (JavaScript). The key factors in this decision were:

* Antlr supports generating JavaScript (and experimental TypeScript) parsing and visitor code.
* VS Code extension authoring is written in TypeScript.
* NodeJS and Browser environments are cross-platform and widely available.

\*\*MORE\*\*

# Implementation

This section describes the development environment, tools and programming languages used to develop Hoshie Lang.

## Development Environment

The following is a description of the development environment used, given the nature of the project setting up an optimised environment served as valuable research as well as expediting actual development.

The following development tools were used:

* **NodeJS**: Cross-platform JavaScript runtime environment for executing JavaScript outside of a web browser. Has a large ecosystem of third-party libraries and a popular package manager to define versioned dependencies - <https://nodejs.org/en/>
* **TypeScript**: An extension to the JavaScript language, TypeScript adds static typing and transpiling capabilities making large scale development of JavaScript more robust - <https://www.typescriptlang.org/>
* **Antlr**: A “compiler compiler” which takes Lexer, and grammar definition files and generates parsers, abstract syntax trees with associated walkers and visitors - <https://www.antlr.org/>
* **Mocha**: JavaScript test framework - <https://mochajs.org/>
* **Chai**: A JavaScript assertion library, commonly used in conjunction with Mocha - <https://www.chaijs.com/>
* **faker**: A JavaScript library for generating fake test data - <https://github.com/Marak/Faker.js>
* **rimraf**: Emulated the "rm -rf" command line in JavaScript. Used for running "clean" commands - <https://github.com/isaacs/rimraf>
* **npm-run-all**: Allows sequential and parallel execution of "npm run" commands. Used for "build-all" type commands - <https://github.com/mysticatea/npm-run-all>
* **watch**: Monitors a folder for changes, used to detect when files change so they can be automatically recompiled - <https://github.com/mikeal/watch>
* **pandoc**: A universal document converter, used to convert markdown to Microsoft Word formats (and back again), allow initial report writing to be done in markdown - <https://pandoc.org/>
* **Visual Studio Code**: Development environment - <https://code.visualstudio.com/>, including the following extensions:
  + **ANTLR4 grammar syntax support**: IDE support for Antlr lexer and grammar files - [https://marketplace.visualstudio.com/items?itemName=mike-lischke.Visual Studio Code-antlr4](https://marketplace.visualstudio.com/items?itemName=mike-lischke.vscode-antlr4)
  + **ESLint**: IDE support for TypeScript linting tools - [https://marketplace.visualstudio.com/items?itemName=dbaeumer.Visual Studio Code-eslint](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint)

## Hoshi Lang Parsing

When developing a new programming language, the first hurdle is to be able to read and understand the files that contain the “Hoshie Lang program” as written by the programmer.

This first step will take the structured text documents and parse them into a logical data representation that can then be used in the next step of the overall process. A side effect of this parsing is that any text that cannot be parsed would be considered a “syntax error”. It is worth noting that it is possible to have syntactically correct code which cannot execute due to other errors (referencing a declaration which has not been declared or is the wrong type for example). Syntax errors are the most basic error the programmer can make and can prevent the program from even being parsed completely. It is common for these types of errors to be caught quickly in an IDE (as the user types) and highlighted so the programmer can fix them quickly.

The Hoshie Lang parser was developed using a tool called ANTLR - *“(****AN****other* ***T****ool for* ***L****anguage* ***R****ecognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.”* (ANTLR Homepage, n.d.).

At a high level the ANTLR tool helps abstract the Hoshie Lang language into a formal definition, commonly known as the “Grammar”, from which it will generate the actual code that converts the text into a usable data structure. This data structure is referred to as an “Abstract Syntax Tree” or AST, as the name suggests it is tree shape structure where each node represents part of the program. Further, ANTLR will also create helper functions for navigating and interrogating the AST, in our case these helpers are referred to as “Visitors” and provide a convenient way to traverse and perform actions on each node in the AST.

### Hoshie Lang Grammar

The formal definition of the Hoshie Lang Grammar is written in the ANTLR “meta-language” and are broken into two distinct parts:

* Lexer
* Parser

This is akin to how a human might read a book, humans don’t read a book one character at a time, but rather they recognise the individual words and process them as a stream that forms a sentence. The meaning of the sentence is then derived from the order and relationship of the words.

The process of grouping the individual characters and symbols into “Tokens” is known as Lexical Analyses (or tokenizing), in ANTLR this is referred to as the “Lexer”. The syntax structure for ANTLR lexers can be found at:  
<https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md> (Lexer Rules, n.d.).

The second part of the process is the “Parser”, the parser takes the stream of tokens and organises it into a tree structure, commonly known as a “parse tree” or “abstract syntax tree” (AST). The syntax structure for ANTLR parsers can be found at: <https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md> (Parser Rules, n.d.)

The AST data structure for a syntactically correct program will precisely represent the entire program and can be used for many different tasks without having to reparse the original program. For example the AST could be used to:

* Colourise the program in an IDE.
* Reformat the original text in the program to a consistent structure, based on a set of rules (tabs or spaces, “=” versus “ = “ etc.).
* Generate executable code (ASM, c++, JavaScript etc.).
* Generate HTML documentation.

#### Hoshie Lang Lexer Definition

The source code the Hoshie Lang Lexer can be located in the sources at “grammar/HLLexer.g4”. It consists of the following main sections (all examples are a small section of the actual code):

* Symbols: A named list of valid symbols, for example:

Comma: ',';

Assign: '=';

Not: '!';

Multiply: '\*';

* Keywords: A named list of keywords, for example:

Boolean: 'boolean';

Number: 'number';

String: 'string';

Return: 'return';

* Functions: Built-in function names:

Length: 'length';

Generate: 'generate';

Random: 'random';

UTest: 'utest';

* Literals: Covers literal values, like boolean, numbers, strings etc.:

NullLiteral: 'null';

BooleanLiteral: 'true' | 'false';

DecimalLiteral

  : DecimalIntegerLiteral '.' DecimalDigit\* ExponentPart?

  | '.' DecimalDigit+ ExponentPart?

  | DecimalIntegerLiteral ExponentPart?

  ;

fragment DecimalDigit: [0-9];

fragment DecimalIntegerLiteral: '0' | [1-9] DecimalDigit\*;

* Identifiers: Hoshie Lang identifiers (must start with a character or “\_” and may consist of characters, numbers and “\_”:

Identifier: IdentifierStart IdentifierPart\*;

fragment IdentifierPart

  : [\_]

  | [a-zA-Z]

  | [0-9]

  ;

fragment IdentifierStart: [\_] | [a-z];

* Strings: Single, double quote strings and escape character definitions
* Other: Whitespace and comment markers

#### Hoshie Lang Parser Definition

The source code the Hoshie Lang Parser can be located in the sources at “grammar/HLParser.g4”. The entire grammar is complex and heavily interlinked, but some of the highlights are:

* Entry point – the root node of the AST:

program: fileElements? EOF;

* File Statement – the main parts of each file:

fileStatement

  : block

  | typeStatement

  | variableStatement

  | actionStatement

  | importStatement

  | exportStatement

  | emptyStatement

  ;

* Assignments – typically assigning an identifier to an expression or function:

variableStatement: variableDeclaration eos;

variableDeclaration: Identifier variableInitialiser;

variableInitialiser

  : '=' singleExpression (As singleTypeExpression)?

  ;

* Single Expressions – these appear in many places, for example in function calls, on the right hand side of expressions and in equality statements:

singleExpression

  : singleExpression arguments

  | '!' singleExpression

  | singleExpression ('\*' | '/' | '%') singleExpression

  | singleExpression ('+' | '-') singleExpression

  | singleExpression ('<' | '>' | '<=' | '>=') singleExpression

  | singleExpression ('==' | '!=') singleExpression

  | singleExpression ('&&' | '||') singleExpression

  | identifier

  | literal

  | arrayLiteral

  | arrowFunction

  | keyword arguments

  ;

#### ANTLR Command Line Tool

The ANTL command line tool is used to convert the lexer and grammar files into actual source code which can be used within our application. Specifically, it is used to generate the following key pieces:

* Lexer / Tokenizer
* Parser
* Error Listener
* Visitor

The command line used for Hoshie Lang:

antlr4  
 -Dlanguage=JavaScript   
 -o ./src/hlcc   
 -lib ./src/hlcc/grammar   
 -visitor   
 ./grammar/HLLexer.g4 ./grammar/HLParser.g4

Where:

* -Dlanguage: Specifies which programming language should be targeted for the output, in this case JavaScript.
* -o: Specifies output directory where all output is generated.
* -lib: Specifies location of grammars, tokens files.
* -visitor: Enable generation of parse tree visitor
* <input files>

### Creating the Abstract Syntax Tree

Having generated the four key pieces, they can now be integrated into our application and be used to parse Hoshie Lang files. The pseudo code to convert the Hoshie Lang text file into an Abstract Syntax Tree is:

charStream = InputStream(“Hoshie Lang Program”)

lexer = HLLexer(charStream)

tokens = CommonTokenStream(lexer)

parser = HLParser(tokens)

ast = parser.program()

1. Create a character stream from the hoshie lang text or file.
2. Create a new Hoshi Lang lexer instance from the character stream.
3. Convert the lexer to a stream of tokens.
4. Create a new instance of the parser from the token stream.
5. Get a reference to the root node in the abstract syntax tree.

The actual code can be viewed in: src/hlcc/parser.ts

### Navigating the Abstract Syntax Tree

\*\*MORE\*\*

## Integrated Development Environment

\*\*MORE\*\*

A critical part of any programming language is good support for associated development tools, VS Code is currently one of the most popular cross platform, general purpose programming IDEs available. https://code.visualstudio.com/

### Contribution Points

VS Code has many ways it can be extended to support new features and languages. These capabilities are defined using “Contribution Points” within the extensions **package.json** file, in a specific section called **contributes**. https://code.visualstudio.com/api/references/contribution-points

* **languages**: Every extension will need to declare some global information about itself, this includes:
  + **id**: A unique ID associated with the extension (hoshie)
  + **extensions**: File extension IDs that VS Code can use to associate a file with the extension (.ho)
  + **configuration**: Some general rules for the language to help the editor handle common operations (./configuration.json) and includes - [link](https://code.visualstudio.com/api/language-extensions/language-configuration-guide):
    - Comment toggling
    - Bracket’s definition
    - Auto closing
    - Auto surrounding
    - Folding
    - Word pattern
    - Indentation Rules
* **grammars**: VS Code uses [TextMate grammar definitions](https://macromates.com/manual/en/language_grammars) for its default syntax colouring. This contribution point associates hoshie lang with a Text Mate language file **./syntaxes/hoshie.tmLanguage.json**.
* **commands**: scasc

# Evaluation

\*\*MORE\*\*

# Conclusions

\*\*MORE\*\*

# Appendix

# References

* Grammar naming conventions and layout based on:
  + https://github.com/antlr/grammars-v4/blob/master/javascript/ecmascript/JavaScript/ECMAScript.g4
  + https://github.com/antlr/grammars-v4/blob/master/javascript/javascript/JavaScriptParser.g4
  + https://github.com/antlr/grammars-v4/blob/master/javascript/javascript/JavaScriptLexer.g4
* Visual Studio Code Syntax Language file bases on
  + https://github.com/microsoft/Visual Studio Code/blob/master/extensions/javascript/syntaxes/JavaScript.tmLanguage.json

# References

*ANTLR Homepage*. (n.d.). Retrieved from https://www.antlr.org/

*Lexer Rules*. (n.d.). Retrieved from https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

*Most Popular Development Environments*. (2019). Retrieved from https://insights.stackoverflow.com/survey/2019#technology-\_-most-popular-development-environments

*Parser Rules*. (n.d.). Retrieved from https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md