Hoshie Lang

*Alexander Smith*

Final-Year Project – BSc in Computer Science

Supervisor: XXXXX

Department of Computer Science

University College Cork

April 2021

# Abstract

Hoshie Lang is a new domain specific programming language focusing on the task of data processing and analysis. It uses the analogy of a data “pipeline”, where rows of data flow through a pipe which contains a series of “activities”, to manipulate the data and “sensors”, to observe the data thus allowing the programmer to focus on the “what” without being concerned with the “how”. It includes a custom extension for the popular Visual Studio Code programming environment and has been architected to support many target platforms.

# Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

* this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
* with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
* with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: .

Date: .

# Acknowledgements

I would like to thank my parents for all their general help and support especially during the pandemic as well as assisting as proof readers, also Bradley Aherne who (with my parents) were involved in many “coffee machine” discussions as the ideas and concepts were initially formalised.

Table of Contents

[Abstract i](#_Toc70251080)

[Declaration of Originality ii](#_Toc70251081)

[Acknowledgements iii](#_Toc70251082)

[Introduction 1](#_Toc70251083)

[Analysis 5](#_Toc70251084)

[Design 9](#_Toc70251085)

[Hoshi Lang Syntax 10](#_Toc70251086)

[Type of Language 10](#_Toc70251087)

[Scoping 10](#_Toc70251088)

[Types 11](#_Toc70251089)

[Expressions 11](#_Toc70251090)

[Functions 11](#_Toc70251091)

[Declarations 12](#_Toc70251092)

[Actions 12](#_Toc70251093)

[Keywords 12](#_Toc70251094)

[Compiler 13](#_Toc70251095)

[Code Generator 13](#_Toc70251096)

[Visual Studio Code Hoshie Lang Extension 13](#_Toc70251097)

[Testing 14](#_Toc70251098)

[Design Conclusion 14](#_Toc70251099)

[Implementation 15](#_Toc70251100)

[Development Environment 15](#_Toc70251101)

[Hoshi Lang Parsing 16](#_Toc70251102)

[Hoshie Lang Grammar 17](#_Toc70251103)

[Hoshi Lang Keywords 20](#_Toc70251104)

[Creating the Abstract Syntax Tree 23](#_Toc70251105)

[Navigating the Abstract Syntax Tree 23](#_Toc70251106)

[Concrete Syntax Tree 24](#_Toc70251107)

[Code Generation 27](#_Toc70251108)

[Integrated Development Environment 28](#_Toc70251109)

[Contribution Points 30](#_Toc70251110)

[Evaluation 31](#_Toc70251111)

[Conclusions 31](#_Toc70251112)

[Appendix 31](#_Toc70251113)

[References 31](#_Toc70251114)

[References 32](#_Toc70251115)

# 

# Introduction

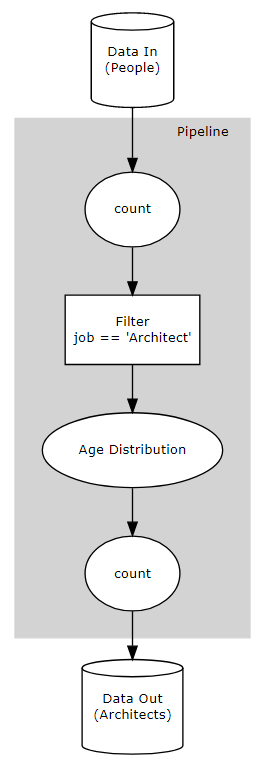
The overall goal of “Hoshie Lang” was to create a new programming language that focused on the task of Data Processing.

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.



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, DISTRIBUTION).

Figure 1

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. While targeting several platforms was beyond the scope of this project, it was still considered to be an important part of the general architecture.

Finally, with any new language the programmer will expect a nice programming environment or integrated development environment. Modern features like syntax checking, problem reporting / highlighting, tooltips and document outlines can greatly assist in a languages use and adoption.

In summary this project creates a new programming language, with support for a modern development experience, with a well encapsulated code generator targeting Web Browsers and Node JS environments.

Simple Example:

/\*

    Calculate age distribution for all Architects.

\*/

Person = {

    string fname,

    string lname,

    number age,

    string job,

    string zipcode,

    string state

};

inCount = count();

outCount = count();

isArchitect = (Person row) => row.job == "Architect";

ageDistribution = distribution((Person row) => row.age);

myProcess = pipeline(inCount, filter(isArchitect), ageDistribution, outCount);

data = readJson("./people.json");

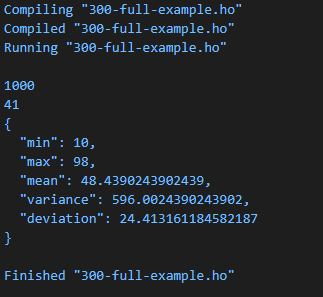
writeJson(myProcess(data), "./people-out.json");

inCount;

outCount;

ageDistribution;

Sample 1 – Simple Hoshie Lang Program  
Declares a named type “Person”  
Declares two count sensors, “inCount” + “outCount";  
Declares predicate function “isArchitect”  
Declares distribution sensor “ageDistribution”  
Declares “myProcess” pipeline with one filter activity and 3 sensors  
Declares “data” as the content of a file (which contains 1000 rows)  
Writes the output of “myProcess(data)” to a file (the filtered data where job == “Architect”)  
Outputs the contents of “inCount”, “outCount" and “ageDistribution” to the console



Output 1 – console output from Sample 1

# 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:

1. Software Developers
2. Data Scientists
3. Database Administrators
4. Computer System Operations
5. Office Workers

These same targets could be further categorised into general competencies:

1. Command Line Users
2. Scripting
3. Programming
4. Querying (SQL)
5. Integrated Development Environment Users
6. Application users
7. Linux
8. Windows
9. OSX

After further discussions, the following (somewhat 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:

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

To appeal to as wide an audience as possible support for a cross platform Integrated Development Environment (IDE) was going to be necessary, the following short list was taken from the 2019 Stack Overflow Survey [1]:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **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” [2] 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 |

The following summary of differences between Antlr and Flex / Bison from Strumenta [3] provided a clear comparison that helped finalise the decision to pick Antlr:

* Stability and Development of New Features
  + flex and Bison are stable and maintained software but there is no active development. C++ support can be of limited quality.
  + ANTLR is actively developed and new features are added periodically
* Separation between Grammar and Code
  + flex and Bison maintain an old-school design with little support for readability or productivity.
  + ANTLR is a modern parsing generator tool with a design that favours a portable grammar, usable for multiple target languages
* Unicode Support
  + flex does not directly support Unicode
  + ANTLR does support all flavours of Unicode and even makes easy to select Unicode properties (e.g., Emoji)
* They Use Different Licenses
  + flex uses a BSD license, while Bison uses the GPL. Bison waive the license for most generated parsers, but it can be a problem
  + ANTLR uses the BSD license
* Grammar Format
  + Bison only supports BNF, which makes grammars more complicated
  + ANTLR supports EBNF, which makes easier to describe most languages
* Features of Lexing Algorithms
  + flex supports regular expressions to define rules, which works for most elements, but adds complexity
  + ANTLR supports context-free expression to define rules, which simplifies defining some common elements
* Features of parsing algorithms
  + Bison supports two parsing algorithms that cover all ranges of performance and languages. It gives cryptic error messages
  + ANTLR supports one algorithm that works for all languages with usually a great performance
* Community and documentation
  + flex and Bison have smaller and fractured communities, but they have good documentation
  + ANTLR has one vibrant community and a good documentation

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 open-source language which builds on JavaScript, one of the world’s most used tools, by adding static type definitions.” [4]

“TypeScript code is transformed into JavaScript code via the TypeScript compiler. This JavaScript is clean, simple code which runs anywhere JavaScript runs: In a browser, on Node.JS or in your apps.” [4]

After further research I satisfied myself that learning TypeScript would be relatively straight forward:

“Adopting TypeScript is not a binary choice, you can start by annotating existing JavaScript with JSDoc, then switch a few files to be checked by TypeScript and over time prepare your codebase to convert completely.”[4]

The decision was made to use **TypeScript** as the primary development language.

At this point picking a suitable **Target** for Hoshie Lang to generate inside its code generator was an obvious choice, **JavaScript** was picked to not only help keep the project manageable, but also because it was a good choice for small to medium sized data sets. It should be noted that the code generation phase should be well encapsulated to facilitate the addition of other targets in the future.

# Design

The mile high goal is relatively simple:

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

Breaking it down further:

1. Design Hoshi Lang Syntax
2. Specify Hoshi Lang Syntax in ANTLR:
   * Lexer
   * Parser
3. Transform output of ANTLR to something useful (Abstract Syntax Tree -> Concrete Syntax Tree)
4. Code Generator (Concrete Syntax Tree -> JavaScript)
5. Visual Studio Code Hoshie Lang Extension (Concrete Syntax Tree -> Extension hooks)

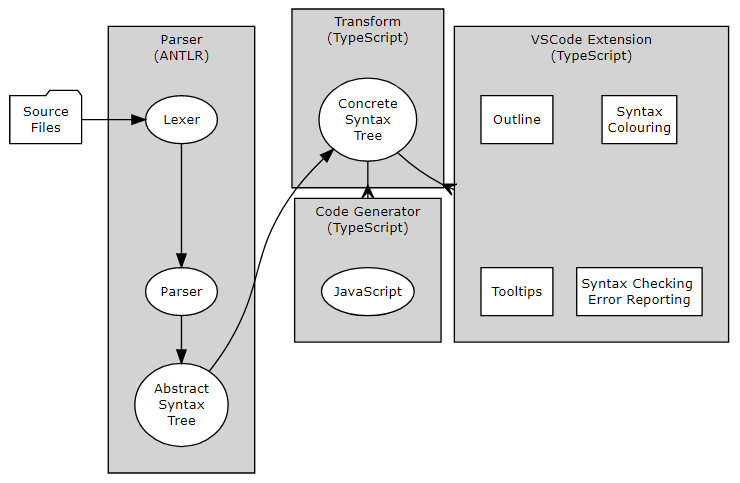


Figure 2 – Design Overview

## Hoshi Lang Syntax

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

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

### Type of Language

From very early on the decision to make Hoshie Lang a **declarative[[1]](#footnote-1)** 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 been 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:

* Always return the same result when the input parameters are the same.
* 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. Explicitly exported declarations from one file can be explicitly imported into another file.
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.

### Types

Hoshie Lang data types fall into two categories:

1. Built in primitives:   
   boolean, number, string
2. User defined “complex” types: These are data structures (like a row in a database):   
   { boolean student, sting name, number age }

Both of these types can be modified with a postpended [] to declare them as “Array of”:

* boolean[]
* { boolean student, sting name, number age }[]

The user can also declare named types. The named type identifier must start with a capital letter:

Person = {boolean student, sting name, number age }

### Expressions

Statements which resolved to a data type:

* Boolean: true, false, equality, greater, less, and, or, not etc.
* Numeric: Plus, minus, multiplication, division, etc
* String: Concatenation

### Functions

Pure functions can be declared and have the following properties:

* Gated inputs (parameters): Typed parameters with optional default values.
* Single return value.
* Two forms (the following are equivalent):
  + Short: (number a, number b = 10) => a + b;
  + Long:  
    (number a, number b = 10) => {  
     // more expresions here...  
     return a + b;  
    };

### Declarations

Named expressions of the form id = “expression”; these should be read as “id is equivalent to expression” and the identifier must start with a lowercase letter.

### Actions

An action is simply an expression statement on its own. Actions are executed in the same order that they are declared in the file:

“Hello World”;

1 + 2;

add(40, 2);

A file with no actions will do nothing when executed.

### Keywords

These are a set of built-in functions and are broken up into the following categories:

1. Pipelines: Defines a list of activities and sensors that will be applied to the data in order. Note: A pipeline is also an activity (see below).
2. Activities: Functions which transform either a data row into a new format or reduce / expand the content of an array.
3. Sensors: Observes data rows as they flow through a pipeline. Note while the sensor declaration is immutable, its internal state is not.
4. File Access: Read and write files to / from disk.
5. Other: General utility and testing functions

## Compiler

The design of the compilation was broken into two phases:

1. The ANTLR generated tokenizer, parser and visitor. The tokenizer and parser are used to create the Abstract Syntax Tree (AST) and the visitor is used to traverse the AST
2. Transformation of the AST to a Concrete Syntax Tree (CST). The CST is a custom data structure which represents the entire Hoshie Lang program, it will consist of several object orientated classes which represent the various “moving parts” of the language. Once instantiated it will have many methods to make navigating and interrogating the structure intuitive and relatively easily, by the code generator and the Visual Studio Code Extension.

## Code Generator

Takes a CST instance and generates code that can be executed. Was designed is such a way that it could target different platforms relatively easily. The included generator targets JavaScript and for reference it is approximately 300 lines of code.

## Visual Studio Code Hoshie Lang Extension

VS Code has a well-documented API [5] for adding support for new languages to the IDE. It should be noted that the total list of possible integrations are very extensive and could serve as an entire project in its own right, some key features were identified and ear marked for implementation:

* Syntax colouring.
* Syntax checking as the user types.
* Error highlighting and reporting as the user types.
* Tooltip information when hovering over words in the Hoshie Lang file.
* Provide overviews for each Hoshie Lang file.
* Enable tasks to compile and execute Hoshie Lang files.

### Testing

Tests were broken into two categories:

1. Unit testing the source code: Traditional unit testing using established testing frameworks.
2. Testing the Hoshi Lang language: By adding a “utest” keyword to Hoshie Lang and a “test” command line option, it will allow continuous testing while the language is developed. It should also serve as a quick way to spot regressions if / when they happen.

## Design Conclusion

The overall design clearly breaks the overall “problem” into four distinct technological challenges:

1. Hoshie Lang design and ANTLR Lexer and Grammar definitions:
   1. Design Hoshie Lang Syntax.
   2. Learn ANTLR and define the syntax.
   3. Create more detailed Hoshie Lang code samples and improve the language specification (back to step “a”).
2. Creating the Concrete Syntax Tree from the ANTLR generated Abstract Syntax Tree:
   1. This is the core port of the project, code generation and IDE integration will be reliant on an accurate representation of the Hoshi Lang program and being able to easily access its content.
   2. Will be written in TypeScript.
3. Generating code that can be executed:
   1. Will generate JavaScript.
   2. Should be capable of generating other languages.
   3. Will be written in TypeScript.
4. Extending VS Code to support Hoshie Lang:
   1. Learn the VS Code extension API.
   2. Learn TextMate specification.
   3. Add hooks for key VS Code features.
   4. Witten in TypeScript.

# 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 [6]
* **TypeScript**: An extension to the JavaScript language, TypeScript adds static typing and transpiling capabilities making large scale development of JavaScript more robust [4]
* **ANTLR**: A “compiler compiler” which takes Lexer, and grammar definition files and generates parsers, abstract syntax trees with associated walkers and visitors [7]
* **Mocha**: JavaScript test framework [8]
* **Chai**: A JavaScript assertion library, commonly used in conjunction with Mocha [9]
* **faker**: A JavaScript library for generating fake test data [10]
* **rimraf**: Emulates the "rm -rf" command line in JavaScript. Used for running "clean" commands [11]
* **npm-run-all**: Allows sequential and parallel execution of "npm run" commands. Used for "build-all" type commands [12]
* **vsce**: The Visual Studio Code Extension Manager [13]
* **watch**: Monitors a folder for changes, used to detect when the grammar files changed so that the ANTLR command line could automatically regenerate the JavaScript lexer, parser and visitor [14]
* **pandoc**: A universal document converter, used to convert markdown to Microsoft Word formats (and back again), allowed initial report writing to be done in markdown [15]
* **yargs**: Helps build interactive command line tools, by parsing arguments and generating an elegant user interface [16]
* **@hpcc-js/dataflow**: A small functional library for processing "data flows" in JavaScript, primarily used in code generation [17]
* **@hpcc-js/util**: Package of JavaScript utilities, primarily used for logging [18]
* **Visual Studio Code**: Development environment [19], including the following extensions:
  + **ANTLR4 grammar syntax support**: IDE support for Antlr lexer and grammar files [20]
  + **ESLint**: IDE support for TypeScript linting tools [21]

## 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.” [7]

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> [22].

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> [23]

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. In our implementation we transform the AST into a Concrete Syntax Tree (CST) which is a more Hoshi Lang specific data structure, with several helper functions for interrogating and navigating the program, once instantiated it is used by:

* Code Generation.
* The IDE integration.

#### Hoshie Lang Lexer Definition

The source code for 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.:

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, for example:

MultiLineComment: '/\*' .\*? '\*/' -> channel ( HIDDEN );

SingleLineComment: '//' ~ [\r\n]\* -> channel ( HIDDEN );

#### 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

  ;

### Hoshi Lang Keywords

The “Keywords” are simply the list of built-in functions and modifiers within Hoshi Lang, these ultimately dictate how easy it will be for the programmer to organise their program and manipulate and analyse their data.

* General:
  + export: Mark a declaration as being “public” outside of the current file –

export hw = "Hello World!";

* + import: Imports specific declarations from another file (must have been explicitly exported)
  + from: Part of the import syntax

import { hw } from "./some-file";

* + return: Returns result from long form function

add2 = (number a, number b) => {

    return a + b;

};

* + typeof: Explicitly types a data structure:

{true, 123, "Hello World"} typeof DataType;

* File:
  + readJson: Reads a JSON file from disk –

data = readJson("./people.json");

* + writeJson: Writes a data as JSON to disk –

writeJson(myProcess(data), "./people-out.json");

* Data Processing:
  + pipeline: Used to specify a series of Activities and Sensors, which the data should flow through (pipeline declarations are also a valid Activity, so can be reused in other pipelines)

myProcess = pipeline(skipN(3), firstN(3));

myProcess2 = pipeline(myProcess, firstN(1));

* Data Processing Activities:
  + filter: Filters rows of data based on provided predicate function.
  + firstN: Allows first N rows of data to pass
  + group: Groups rows of data based on provided function
  + groupCount: Similar to group but replaces the grouped rows with a count of those rows (convenience)
  + map: Transforms one data type to another data type (by mapping fields)
  + skipN: Skips N rows of data
  + sort: Sort data based on provided predicate function
* Data Processing Sensors:
  + count: Counts the rows
  + deviation: Calculates the standard deviation of the value returned from provided function
  + distribution: Calculates the min, mean, max, deviation and variance of the value returned from provided function
  + extent: Calculates the min and max value of the value returned from provided function
  + max: Calculates the maximum value returned from provided function
  + mean: Calculates the mean value returned from provided function
  + median: Calculates the median value returned from provided function
  + min: Calculates the minimum value returned from provided function
  + quartile: Calculates the quartiles for the value returned from provided function
  + variance: Calculates the variance for the value returned from provided function
* Testing:
  + utest: Built-in unit testing, only executed when user specifies the “test” option.

utest(5 + 7, 12);

utest(15 - 3, 12);

utest(10 > 5, true);

utest(add(1, 2), 3);

It is worth noting that “utest” was implemented very early in the project, which greatly helped to expedite the language development.

#### 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

ANTLR also has the capability to generate two types of AST navigation aids:

* Listeners
* Visitors

For Hoshie Lang we used the Visitor generated code. This is essentially an Object Orientated base class which has a “visit” method for each declaration in the grammar file. Each one of these “visit” methods is passed a “context” state object which provides the following information:

* Location information (file position, line number column number etc.)
* The original text that matched the grammar declaration.
* Helper methods to extract the different parts of the declaration.

Further, the Visitor base class has methods to walk the AST structure and each visit method can return a custom object which will propagate back up the calling hierarchy. The following example shows how to process an “AdditiveExpression” (a + b or a - b):

visitAdditiveExpression(ctx) {

  const [lhs, , rhs] = super.visitAdditiveExpression(ctx);

  if (lhs.type === "number" && rhs.type === "number") {

  } else if (lhs.type === "string" && rhs.type === "string" && ctx.Plus(){

  } else {

    this.ctxError(ctx, "Additive Expression is not valid");

  }

  return new AdditiveExpression(ctx,this,lhs,rhs,ctx.Plus() ? "+" : "-");

}

In this example:

1. Call the base class method to visit the children and gather up the Hoshie Lang Concrete Objects created by same (lhs and rhs).
2. Additional error checking: 1+1, 1-1 is valid and “aaa” + “bbb” is valid (string concatenation), while “aaa” - “bbb” (and all other combinations) are not valid.
3. Return a Hoshie Lang Concrete Object representing the additive expression.

### Concrete Syntax Tree

The primary reason to create the “Concrete Syntax Tree”, is that it can contain lots of specific helper methods to assist with downstream functionality like code generation and IDE integration.

The data structure itself was implemented as a Class Hierarchy:

* XXX
* HLNode: Base class with helpers to resolve file positions and methods to format errors.
  + HLScope: Common helpers for “scopes” of code, contains lists of declarations, actions, tests, errors and types
    - HLFunctionScope: Adds Function specific helpers, for parameters and return types.
    - HLFileScope: Adds File specific helpers, primarily for import and export of declarations.
  + HLDeclaration
    - Declaration: An Identifier and expression pair
    - Alias: An alias for an identified (used in imports)
  + HLExpression: Base class for all expressions exposes abstract helpers to resolve the “type” of the expression and an “eval” function to evaluate the value (when possible), the following derived classes augment the base class:
    - NotExpression: NOTs boolean expressions
    - MultiplicativeExpression: Multiplies or divides two numeric expressions
    - AdditiveExpression: Adds, subtracts two numeric expressions or concatenates two string expressions
    - RelationExpression: Compares two numeric or string expressions for <, <=, >, >=
    - EqualityExpression: Compares two expressions for equality
    - LogicalExpression: AND, OR for two boolean expressions
    - IdentifierExpression: An ID referencing another previously declaration
    - BooleanExpression: true or false literal
    - NumericExpression: numeric literal
    - StringExpression: string literal
    - DataExpression: complex data row literal
    - ArrayExpression: Array of literal expression
  + HLFunction: Base class for built-in functions, activities and sensors
    - LengthFunction: Calculates length of an array or string
    - RandomFunction: Generates random number
    - GenerateFunction: Generates N items determined by provided function
    - ReadJsonFunction: Reads json from file conceptually as a stream of data
    - WriteJsonFunction: Writes a stream of data to a file
    - ActivityFunction: Functions that can alter the data that passes through a pipeline
      * FilterFunction: Filters data based on predicate function[[2]](#footnote-2)
      * FirstNFunction: Allows N rows to pass through
      * GroupFunction: Groups data based on an accessor function[[3]](#footnote-3)
      * GroupCountFunction: Groups data based on an accessor function and then counts the number of rows associated with that group
      * MapFunction: Maps one row of data to a new “shape” of data
      * PipelineFunction: Streams data through a list of activities and sensors. Pipelines are also activities to encourage reuse.
      * SkipNFunction: Skips N rows and allows the rest to flow through the pipeline.
      * SortFunction: Sorts the data based on a predicate function.
    - SensorFunction:
      * CountFunction: Counts the number of rows that pass through the sensor.
      * DeviationFunction: Calculates the standard deviation[[4]](#footnote-4) for a value returned from an accessor function.
      * DistributionFunction: Calculates the min, max, mean, standard deviation and variance for a value returned from an accessor function.
      * ExtentFunction: Calculates the min and max value returned from an accessor function.
      * MeanFunction: Calculates the mean value returned from an accessor function.
      * MedianFunction: Calculates the median value returned from an accessor function.
      * MaxFunction: Calculates the max value returned from an accessor function.
      * MinFunction: Calculates the min value returned from an accessor function.
      * QuartileFunction: Calculates the quartiles[[5]](#footnote-5) for the value returned from an accessor function.
      * VarianceFunction: Calculates the variance[[6]](#footnote-6) for the value returned from an accessor function.
  + HLAction:
    - InlineAction: A single expression with no declaration.
    - Test: Built-in unit test function “utest”. Compares two expressions and asserts that they are equal. Only executes when the "--test" option is passed to the command line interface.

## Code Generation

At the highest level the code generation process takes an instance of a Concrete Syntax Tree (CST) and converts it to source code that can be executed in the target environment. As the generated code should be optimal for the environment, this phase should be overseen by a domain expert.

This project includes a JavaScript code generator, which leverages the @hpcc-js/dataflow [17] JavaScript library for the data processing steps, as it is a modern data processing library using Generators[[7]](#footnote-7) to efficiently iterate over data structures, it also provides a set of functions that map cleanly to the pipeline paradigm and could be used when generating the activities and sensors, thus providing the domain expertise needed. The rest of the code generation created standard JavaScript code.

As the CST provides a set of methods making the iteration over the actions and the resolving of identifiers to expressions, the code generation process is a relatively straightforward process (pseudocode):

1. “Create Code” function is called with an instance of a CST.
2. For each action in the CST, we call a method “generate”.
3. “generate” calls a corresponding specific method for each known CST Node Type.
4. These specific methods will output a source code template recursively calling “generate” as needed:

NotExpression(row: NotExpression) {

    return `!${this.generate(row.expression)}`;

}

AdditiveExpression(row: AdditiveExpression) {

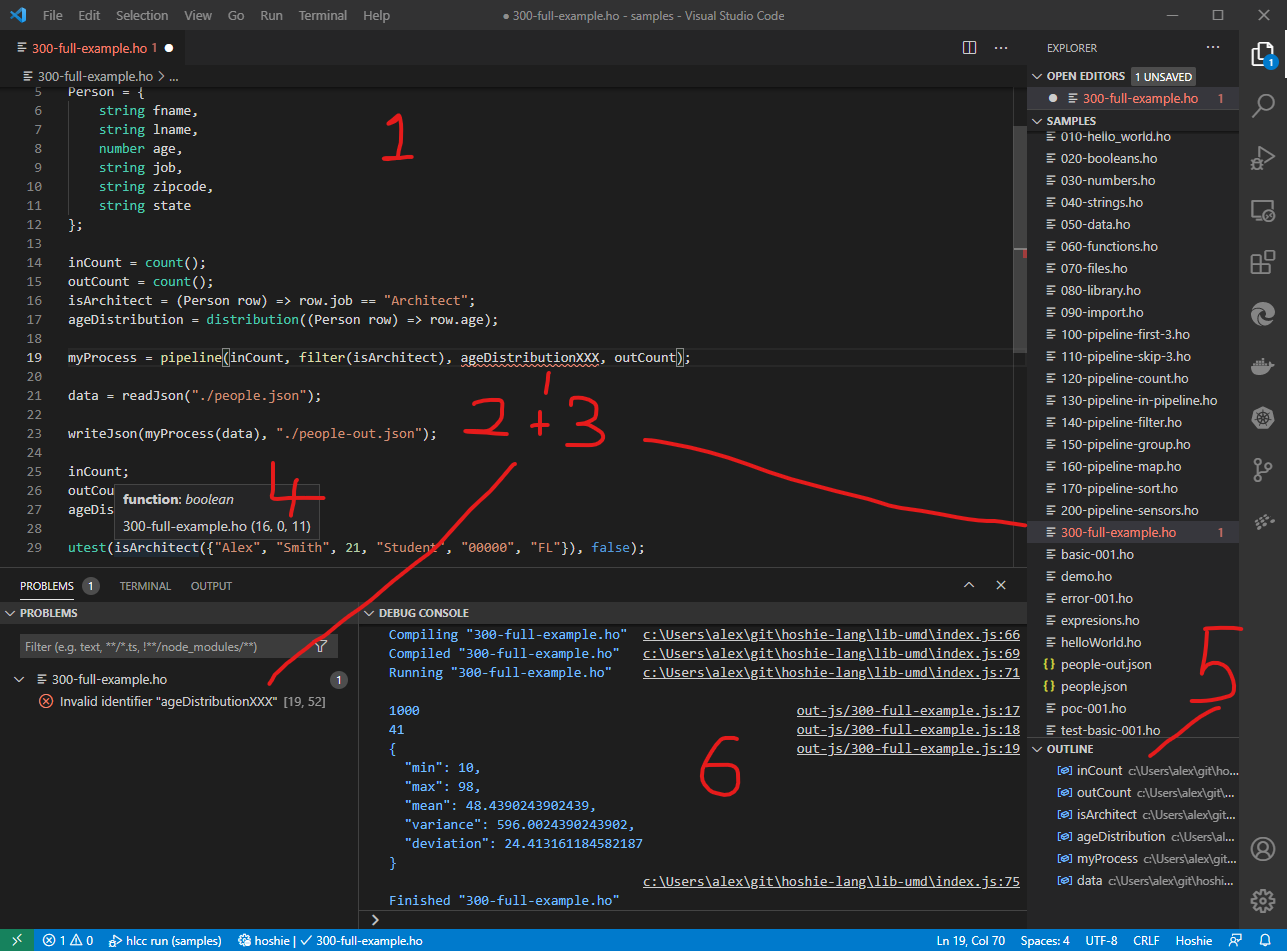
    return `${this.generate(row.lhs)} ${row.action} ${this.generate(row.rhs)}`;

}

## Integrated Development Environment

Extending Visual Studio Code to include support for Hoshie Lang was a more traditional learning experience, it has an extensive API to allow the extension author to “contribute” to the VS Code environment, potentially an entire project could be devoted to this exercise on its own. There were however several contributions that would be essential not only for the Hoshie Lang developer, but also to aid in the development of the Hoshie Lang syntax, grammar and AST -> CST transformations. It is worth noting that developing the extension in tandem with the development of the language syntax and compiler helped expedite the overall process greatly.

Hoshie Lang added the following contributions:

1. Syntax Colouring
2. Syntax Checking
3. Error / Warning reporting
4. Hover (Tooltip) Information
5. Code Outline
6. Execute current file

### 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** [24].

* **main**: The main source file to load when a specified “Activation Event” occurs.
* **activationEvents**: Specific events that signify the Hoshie Lang extension should load. In this case it gets triggered whenever a “.ho” file is opened within VS Code.
* **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:
    - Comment toggling
    - Bracket’s definition
    - Auto closing
    - Auto surrounding
    - Folding
    - Word pattern
    - Indentation Rules
* **grammars**: VS Code uses TextMate grammar definitions [24] for its default syntax colouring. This contribution point associates hoshie lang with a Text Mate language file ./syntaxes/hoshie.tmLanguage.json.
* **commands**: Adds commands to global list and context menus. For example Syntax Check for .ho files.
* **menus**: Associates commands with specific menus
* **keybindings**: Associates commands with hotkeys

After the specified action event triggers the extension to load, it performs the following initializations:

* Command Handlers
* Diagnostic Collection (Syntax Error Reporting)
* Document Symbol Provider (Outline)
* Hover Provider (tooltips)

The handlers and providers work in a similar way:

1. As the developer works in VS Code, they will trigger various callbacks
2. The callbacks will include a reference to the current document, either active of being edited
3. The extension will then create a CST for this document
4. The extension will then use the methods in the CST to provide the information needed (the text in the Hover, or the labels and types in the Symbol Provider for example).

The Diagnostic Collection is slightly different in so far as it can be updated at any time, typically on a requested syntax check, but also on events like file save and even as the user types.

Note: For “live” syntax checking the extension simply reported any errors and warnings at the same time it created the Document Symbols, which is called as the user types, but only on “idle” times.

# Evaluation

# 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](https://github.com/microsoft/Visual%20Studio%20Code/blob/master/extensions/javascript/syntaxes/JavaScript.tmLanguage.json)

# References

|  |  |
| --- | --- |
| [1] | “Most Popular Development Environments,” 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-\_-most-popular-development-environments. |
| [2] | [Online]. Available: https://en.wikipedia.org/wiki/Compiler-compiler. |
| [3] | [Online]. Available: https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/. |
| [4] | [Online]. Available: https://www.typescriptlang.org/. |
| [5] | “Visual Studio Code - Extension API,” [Online]. Available: https://code.visualstudio.com/api. |
| [6] | [Online]. Available: https://nodejs.org/en/. |
| [7] | “ANTLR Homepage,” [Online]. Available: https://www.antlr.org/. |
| [8] | [Online]. Available: https://mochajs.org/. |
| [9] | [Online]. Available: https://www.chaijs.com/. |
| [10] | [Online]. Available: https://github.com/Marak/Faker.js. |
| [11] | [Online]. Available: https://github.com/isaacs/rimraf. |
| [12] | [Online]. Available: https://github.com/mysticatea/npm-run-all. |
| [13] | [Online]. Available: https://github.com/microsoft/vscode-vsce. |
| [14] | [Online]. Available: https://github.com/mikeal/watch. |
| [15] | [Online]. Available: https://pandoc.org/. |
| [16] | [Online]. Available: https://yargs.js.org/. |
| [17] | [Online]. Available: https://github.com/hpcc-systems/Visualization/tree/trunk/packages/dataflow. |
| [18] | [Online]. Available: https://github.com/hpcc-systems/Visualization/tree/trunk/packages/util. |
| [19] | [Online]. Available: https://code.visualstudio.com/. |
| [20] | [Online]. Available: https://marketplace.visualstudio.com/items?itemName=mike-lischke.Visual Studio Code-antlr4. |
| [21] | [Online]. Available: https://marketplace.visualstudio.com/items?itemName=dbaeumer.Visual Studio Code-eslint. |
| [22] | “Lexer Rules,” [Online]. Available: https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md. |
| [23] | “Parser Rules,” [Online]. Available: https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md. |
| [24] | [Online]. Available: https://macromates.com/manual/en/language\_grammars. |
| [25] | J. Lloyd, “Practical Advantages of Declarative Programming”. |
| [26] | J. Bland and D. Altman, in *Statistics notes: measurement error*, 1996. |
| [27] | [Online]. Available: https://en.wikipedia.org/wiki/Quartile. |
| [28] | [Online]. Available: https://en.wikipedia.org/wiki/Variance. |

1. Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow [25] [↑](#footnote-ref-1)
2. A predicate function is one that resolves to true or false from a row of data [↑](#footnote-ref-2)
3. An accessor function is one that returns a single value from a row of data [↑](#footnote-ref-3)
4. The standard deviation is a measure of the amount of variation or dispersion of a set of values [26] [↑](#footnote-ref-4)
5. a quartile is a type of quantile which divides the number of data points into four parts, or quarters, of more-or-less equal size [27] [↑](#footnote-ref-5)
6. variance is the expectation of the squared deviation of a random variable from its mean [28] [↑](#footnote-ref-6)
7. a generator is a routine that can be used to control the iteration behaviour of a loop. All generators are also iterators. A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time, which requires less memory and allows the caller to get started processing the first few values immediately. In short, a generator looks like a function but behaves like an iterator. [29] [↑](#footnote-ref-7)