Custom Language & Compiler

Benjamin Graham

Candidate Number: 1194

Centre Name: The Weald School

Centre Number: 65103

Testing Video:

Link Not Found

A Level Computer Science NEA

AQA

**Contents**

[**Analysis 3**](#_wz2hhrao6sl0)

[Problem 3](#_guybhxv5f241)

[Overview 3](#_48b5mgek0dt6)

[What is a Compiler 3](#_99za88891xp5)

[Third Parties 6](#_fu0x1ig44ekv)

[Research 6](#_g5ccryw66e3e)

[Interviews 6](#_mq48h8a4vfxm)

[Prototypes 6](#_hyb9pemahjfl)

[Objectives 6](#_6jajixp040r5)

[**Design 7**](#_kr6fg9ckdrz6)

[Sub 1 7](#_57p0vm8pgwv)

[**Technical Solution 8**](#_qrw3jsbdpeuu)

[Sub 1 8](#_jn07abohykqh)

[**Testing 9**](#_df6tac3rb3f0)

[Sub 1 9](#_l80njv6vn0y)

[**Evaluation 10**](#_rqh2fxij50qm)

[Sub 1 10](#_qort43mlrah)

[**References 11**](#_7wculjusut4)

[**Appendix (code) 12**](#_cfpbspmrin2v)

[Sub 1 12](#_6zxcavmhh1s)

# 

# 

# Analysis

## Problem

### 

### Overview

Many new programmers start with simpler languages such as Python or Lua as this helps them get a start in programming due to the simple syntaxes that are easy to learn and debug. However, this could make learning the more complex industry-standard languages, such as C++ or Rust difficult, as they have a much more complex syntax and structure. As a result, this could cause these programmers to lose motivation when trying to learn a new language.

As a solution, I want to create a programming language that bridges the gap between simple languages and more complex languages. This will help the new programmers learn the structure of complex languages used in the industry while still keeping the language easy to understand and write.

Many of the more complex languages utilise compilers to translate the source code into machine code. Due to this, I would like to create a compiler for this language as it would help the programmers get used to how compilers function, including how the compiler takes in the code and how the resulting machine code is run.

### What is a Compiler

Compilers differ from interpreters as they return machine code, which is later run, instead of running it line by line. This results in the code being executed faster as it does not need to be translated every time it needs to be run. Furthermore, the machine code is run directly on the computer's processors instead of being run by another program again increasing the efficiency and speed of compiled code.

To translate code the compiler passes the source code (original code ) through multiple different programs, as shown in [Figure 1](#_pwmk8urjwb96), to slowly break down the code into a form that allows machine code to be generated easily.

#### Figure 1: Stages of a compiler

Sourced from “Introduction to Compiler and Language Design; Pg 6“

The Scanner, or Lexical Analyser, Takes in the characters of the source code as a stream, one by one, and groups them into tokens.

Most languages have these types of tokens:

* Keywords
  + These are words reserved by the language itself that build the general structure of the language. (e.g. While, True, If)
* Identifiers
  + The names of functions, variables, or classes in the program. The programmer chooses these and can be anything within a set of rules. Typically these rules are a sequence of letters, digits, or underscores that cannot start with a digit (e.g. NumOfCheese, Player\_1)
* Numbers
  + Whole or fractional numbers, including different bases (binary, hex, octal) (e.g. 2, 2.9, 0b01001, 2e10)
* Strings
  + Sequences of characters that are typically quoted with double or single quotation marks to allow them to be distinguished from Keywords and Identifiers (e.g. “Hello, World!”, ‘You Win’, “WHAT?”)
* Operators
  + Characters used during boolean or arithmetic operations (e.g. +, ^, %)
* Delimiters
  + Characters used to separate parts of the code (e.g. (), {}, ;)

For example the tokens for the program:

Area = Height \* (8 + 2);

Would be:

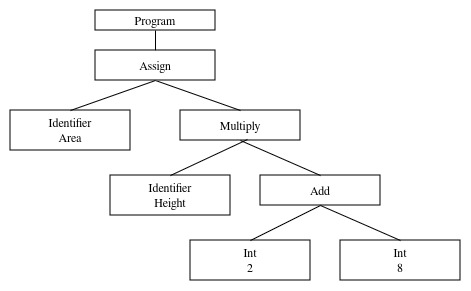
[ID: Area][Op: =][ID: Height][Op: \*][Delim: (][Int: 8][Op: +][Int: 2][Delim: )][Delim: ;]

These tokens are like the words of the programming language. However, if a character cannot be grouped into a token a lexical error will be raised. Some examples of these errors are an integer exceeding the maximum value that can be represented or an illegal character in the code.

The Parser, or Syntactic Anayliser, will take the stream of tokens generated by the Scanner and group them into expressions which are like the sentences of the programming language. This is achieved by following the rules set by the language grammar which show how expressions can be formed with certain tokens.

After grouping the tokens as much as possible the parser will construct an Abstract Syntax Tree (AST). This is a representation of the original code that has the grammatical structure of the program but removes unnecessary details.

Due to how the AST is generated, Each node represents a rule in the language's grammar, and the whole AST shows how each token builds the program.



#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### Figure 2: Simple AST for the example program above

## 

A syntax error will be raised when the tokens cannot be grouped into a complete program. Examples of these include missing commas/brackets or using keywords as an identifier.

The semantic analyser takes the AST generated by the parser and goes through it to extract extra information such as data types for expressions. Some examples of data that it could extract are:

* Type data for expressions
* The Scope of Variables
* Making sure no errors occur (such as typing errors)
* Identifying dead code (code that is not used)

After extracting this extra information the semantic analyser outputs an Intermediate Representation (IR). These are data structures much like the AST but with the extracted information, including:

* AST (with the additional information added)
* Control Flow Graph
* Directed Acyclic Graph

IRs make it possible to add optimisations by editing the code due to the simpler and more abstracted way they represent the program. Optimisations might include removing dead code or combining arithmetic operations which reduces the amount of line of code that needs to be generated/run

The IR will then be passed through a code generator. This generates an assembly language representation of the code, which can then be passed to an assembler to generate the machine code.

To avoid different variables fighting over registers/memory space the code generator will use the extra information extracted from the semantic analyser such as the lifetime of variables to assign registers or allocate/deallocate memory space to the variable.

## Third Parties

## Research

## Interviews

## Prototypes

## Objectives

## 

## 

# Design

## Sub 1

# 

# Technical Solution

## Sub 1

## 

# Testing

## Sub 1

# Evaluation

## Sub 1

# References

| Authors Name | Book/Website Name | Publisher's Name | Year of Publication | URL/ISBN |
| --- | --- | --- | --- | --- |
| PG Online | Tackling A-Level Projects in Computer Science | PG Online Limited | 2020 | 9781910523209 |
| Prof. Douglas Thain | Introduction to Compilers and Language Design (Second edition) | University of Notre Dame | 2023 | 9798655180260 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 

# Appendix (code)

## Sub 1