Custom Language & Compiler

Benjamin Graham

Candidate Number: 1194

Centre Name: The Weald School

Centre Number: 65103

Testing Video:

N/A

A Level Computer Science NEA

AQA

**Contents**

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

[Problem 3](#_guybhxv5f241)

[Overview 3](#_48b5mgek0dt6)

[How compilers work 3](#_99za88891xp5)

[Lexical analyser 4](#_i1206lm8nztm)

[Syntactic analyser 5](#_7l59a3i3k3dm)

[Semantic analyser 6](#_xemsovpkboma)

[Code generator 6](#_u0dwq13ufdl3)

[How are programming languages defined 6](#_bbm1naic07ga)

[Third Parties 7](#_fu0x1ig44ekv)

[End User 7](#_f7yw68d4n23)

[Research 7](#_g5ccryw66e3e)

[Lexical Analysis 7](#_e9497vxb08zk)

[Regular expressions 7](#_ols4wq81t3by)

[Finite Automata 9](#_ewcjr6tfywsa)

[Automatic lexical generators 10](#_94qmkq3wimjk)

[Syntactic analysis 11](#_1djb1p2mnzmz)

[Context-free grammars 11](#_kp6gkqdv001w)

[Objectives 13](#_6jajixp040r5)

[Design 13](#_kr6fg9ckdrz6)

[Sub 1 13](#_57p0vm8pgwv)

[Technical Solution 14](#_qrw3jsbdpeuu)

[Sub 1 14](#_jn07abohykqh)

[Testing 15](#_df6tac3rb3f0)

[Sub 1 15](#_l80njv6vn0y)

[Evaluation 16](#_rqh2fxij50qm)

[Sub 1 16](#_qort43mlrah)

[References 17](#_7wculjusut4)

[**Appendix (code) 18**](#_cfpbspmrin2v)

[Sub 1 18](#_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 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.

### How compilers work

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' no fts 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“

##### Lexical analyser

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
  + Integer or floating point 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: ;]

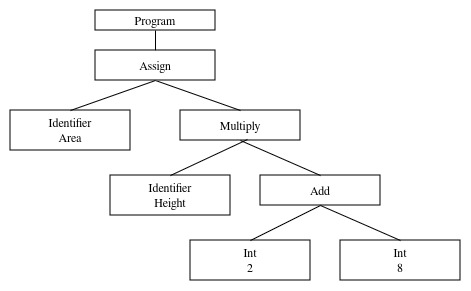
These tokens are like the words of the programming language. However, if a character cannot be grouped into a token a lexical error will be raised. Some examples of these errors are an integer exceeding the maximum value that can be represented or an illegal character in the code.

##### Syntactic analyser

The Parser, or Syntactic Analyser, 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.

##### Semantic analyser

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.

##### Code generator

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 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.

### How are programming languages defined

Before creating a translator for a programming language it needs to be well defined. This involves laying out the grammar of the programming language, which will lay out how the syntax of the language will be written and interacted with by the programmer.

The syntax of the programming language is usually defined using a context-free grammar. These are sets of rules which represent how parts of the language are combined and are used primarily in the syntactic analysis stage of compilation.

## Third Parties

### End User

## Research

### Lexical Analysis

As explained before the lexical analyser takes in the raw stream of characters from the source code and groups them into tokens (tokenisation). Some of the main ways lexical analysers work are:

* Regular expressions
* Finite automata
* Automatic lexical generators

##### Regular expressions

Regular expressions (Regex’s) are a pattern of characters used to match patterns within a string. Usually, the character in the expression has to be matched 1 to 1. Example regular expressions and what they match are listed below:

* Dog -> { “Dog” }
* Cat -> { “Cat” }
* House -> { “House” }

However, some characters have special properties within regular expressions which change the meaning of characters around them. Some of these are:

* ‘( )’
  + Brackets create a capture group which can be used to group characters together to allow another special character to interact with more than one character in the expression
* ‘\*’
  + Asterisks denote that the character or group before it can be repeated 0 or more times
* ‘+’
  + Plusses denote that the character or group before it can be repeated 1 or more times
* ‘?’
  + Question marks denote that the character or group before can appear 1 or 0 times ( In other words the character or group before the question mark is optional )
* ‘|’
  + A vertical bar denotes the expression on the left can be matched OR the expression on the right can be matched
* ‘.’
  + A full stop denotes that any character can be matched in place of it
* ‘[ ]’
  + Square brackets denote that any character contained within them can be matched in place of it
  + If the first character within the square brackets is “^” the other characters are negated and the square brackets can be interpreted as any character but the characters within the brackets
  + A “-” within the square brackets denotes the characters between the two on either side of the Hyphen are included in the group (e.g. a-z denotes any lowercase letter whereas A-Z denotes any uppercase letter)

Some examples of regular expressions with special characters are:

* (Dog|Cat)? -> { “”, “Dog”, “Cat” }
* Moo+ -> { “Moo”, “Mooo”, “Moooo”, … }
* Gr[ae]y -> { “Gray”, “Grey” }

Regular expressions can be used in lexical analysis as they are great at outlining what is allowed in a certain situation, allowing them to effectively state what characters are allowed within a token. For example, these regular expressions can be used to denote some of the tokens listed before:

* (While|For|If|Else|...)
  + Matches all the keywords used in the programming language
* [a-zA-Z\_][a-zA-Z0-9\_]\*
  + Matches all the possible identifiers
* -?[1-9][0-9]\*|0
  + Matches all possible positive and negative integers (excluding non-base 10 numbers)
* (-?[1-9][0-9]\*|0)\.[0-9]+
  + Matches all possible positive and negative decimal floats

Regular expressions also are quite efficient at parsing and matching text meaning they don't increase compilation time too much. One problem with using regular expressions for lexical analysis is that while they are great at compressing patterns into a few characters, with longer more complex patterns regular expressions can get quite long and hard to understand and debug.

##### Finite Automata

Finite automata (FA), like regular expressions, are used to match patterns in strings. However instead of using a written string to denote these patterns they work like a sort of abstract machine which changes ‘state’ when a particular input is seen. The input to transition from one state to another is called an ‘edge’. Furthermore one or more of the states will be an accepting state. This means the input can be accepted as matching the pattern when the automata gets to that state.

Graphically finite automata are usually represented using circles for the states and arrows (with the character needed) for the edges. Accepting states are usually represented as a double circle to allow them to be different from other states.

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### Figure 3: Simple FA to match ‘for’

#### Sourced from “Introduction to Compiler and Language Design; Pg 16“

Another way finite automata can be represented is by using a transition table. This is a simple table which has the states down one axis and the input symbols down the other. The table is then filled in with which states can be reached using the symbol and state of the cell.

|  | f | o | r | ε |
| --- | --- | --- | --- | --- |
| 0 | 1 | / | / | / |
| 1 | / | 2 | / | / |
| 2 | / | / | 3 | / |
| 3 | / | / | / | / |

#### 

#### Figure 4: Transition table for FA shown in Fig. 3

The ‘ε’ represents an ‘epsilon’ transition which matches an empty string. This means that the transition can be taken without taking any input value.

##### Automatic lexical generators

Automatic lexical generators are programs that take in a list of rules, usually described in regular expressions, for each token in your programming language and return a lexical analyser which follows these rules.

One commonly used version of an automatic lexical generator is Flex. Flex takes in a file split into three sections which include: a defining section, a rules section and a subroutine section. These are split in the file using two percentage signs (“%%”).

#### Figure 5: Example flex input file to match integers

In the defining section, variables are defined which can be used later in the code, these could include error strings so you do not have to output them each time, the rules sections include regular expressions followed by c-code which will be run if the given regular expression matches and the subroutine section contains subroutines which can be used in the rules section for when a regular expression matches.

Automatic lexical generators take much less work than making your own “hand-written” lexical analyser however it makes it much less customisable as the outputted code is much harder to access and edit.

### Syntactic analysis

As said above, during the syntactic analysis stage the compiler takes the tokens generated by the lexical analyser and groups them. To do this a context-free grammar is used as the guide to how the tokens should be grouped.

##### Context-free grammars

Context-free grammars are made up of two main parts, Terminals and non-terminals. Terminals are the tokens generated by the lexical analyser and non-terminals are structures made up of terminals and other non-terminals.

As an example, an assignment expression (when a value gets assigned to a variable) would be a non-terminal made up of the terminals “identifier”, “=” and the non-terminal “expression”. Graphically this can be represented as:

<Assignment> -> Identifier = <Expression>

Each rule in the grammar can be represented in this way, with a single non-terminal on the left and a group of terminals and non-terminals on the right.

The non-terminal of the “top” rule of the program is known as the start symbol and any valid group of non-terminals can be derived from it. In most grammars for programming languages, the start symbol would be a “program”.

There are two methods of using the grammar to “prove” a given set of tokens can be derived from the start symbol:

* Top-down derivation
* Bottom-up derivation

In top-down derivation, you begin with the start symbol and apply the rules of the grammar to expand non-terminals until the set of tokens is reached.

In bottom-down derivation, you begin with the group of tokens and apply rules until you reach the start symbol.

Furthermore, there can be multiple possible derivations for a single set of tokens, this is due to which order you apply the rules, and this can sometimes result in different outcomes for parsing. When this is possible the grammar is called ambiguous.

There are many methods to reduce ambiguity in a grammar including:

* Adding extra non-terminals, which are used to break down complex rules into simpler ones or
* Removing left-recursion, left-recursion happens when the non-terminal on the left of the rule is the same as the leftmost symbol of the right-hand side. This can be removed by adding a new rule which handles the recursion while the the old rule just handles the starting term

#### Figure 6: Example of Left-recursion and how it is removed

Just like with Lexical Analysis, multiple methods exist for constructing a parser from a given Context-free Grammar. These methods include:

* Table driven parsing
* Recursive descent parsing

`

To perform most methods of parsing two sets are needed, a *first* and *follow* set. Each non-terminal in a grammar will have these two sets, the first set includes the terminals which could appear at the start of a string derived from the non-terminal and the follow set includes any tokens that appear after the non-terminal in a string.

For example, the first and follow sets for the example grammar in [Figure 6](#_74p4vu8pv1nw) would be:

|  | P | E | E’ | T |
| --- | --- | --- | --- | --- |
| First | ID | ID | ‘+’ ε | ID |
| Follow | $ | $ | $ | ‘+’ $ |

Where ‘$’ matches the end of the file and ‘ε’, like before, matches an empty string

##### Table Driven parsing

In table-driven parsing uses a stack which initially contains the start symbol and a table which contains the rule needed to be used based on the input terminal and non-terminal.

|  | **ID** | **+** | **$** |
| --- | --- | --- | --- |
| **P** | 1 |  |  |
| **E** | 2 |  |  |
| **E`** |  | 3 | 4 |
| **T** | 5 |  |  |

#### 

#### Figure 7: Parsing table for the example grammar

When parsing the top item on the stack and the next input token are used as inputs for the parsing table to get the next rule that can apply, if the top item on the stack is a terminal and it matches the next input token however both will be removed from the stack and input stream.

If a rule cannot be applied or the next item on the stack does not match the next input token a syntax error will be raised.

As an example, the parsing for the String of tokens “ID + ID + ID“ would look like:

| Stack | Input stream | Action |
| --- | --- | --- |
| P $ | ID + ID + ID $ | Perform 1 ( P -> E ) |
| E $ | ID + ID + ID $ | Perform 2 ( E -> T E` ) |
| T E` $ | ID + ID + ID $ | Perform 5 ( T -> ID ) |
| ID E` $ | ID + ID + ID $ | Match ID |
| E` $ | + ID + ID $ | Perform 3 ( E` -> + T E` ) |
| + T E` $ | + ID + ID $ | Match ‘+’ |
| T E` $ | ID + ID $ | Perform 5 ( T -> ID ) |
| ID E` $ | ID + ID $ | Match ID |
| E` $ | + ID $ | Perform 3 ( E` -> + T E` ) |
| + T E` $ | + ID $ | Match ‘+’ |
| T E` $ | ID $ | Perform 5 ( T -> ID ) |
| ID E` $ | ID $ | Match ID |
| E` $ | $ | Perform 4 ( E` -> ε ) |
| $ | $ | Match $ |

##### Recursive Descent Parsing

In Recursive descent parsing a parsing function is made for each non-terminal in the grammar. Each function will recursively call the function associated with the next non-terminal based on the rules of the grammar. This happens until a terminal is reached and found in the input stream or an error is found.

[Figure 8](#_i0uv83uq1ahf) shows a simplified version of what a parsing function might look like for parsing the E` non-terminal in the example grammar above:

#### Figure 8: Recursive descent parsing function example

### Semantic analysis

## Objectives

* Layout keywords and syntax of programming language
  + Keeping the keywords simple and syntax easy to read due to responses from end-user questionnaire
* Create a Lexical analyser using regular expressions or finite-state machines
* Create a Syntactic analyser using a table-driven approach or recursively
* Design a type system for the programming language
* Create a semantic analyser including type checking and scope resolution
* Create a code generator to link with an existing assembler such as NASM
* Write simple built-in programs for the language to create a standard library
  + Keeping them easy to use and straightforward due to responses from end-user questionnaire

#### 

#### 

#### 

# 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 |
| Alfred V. Aho  Monica S. Lam  Ravi Sethi  Jeffery D. Ullman | Compilers  Principles, Techniques, & Tools | Pearson | 1986 | 9780201100884 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 

# Appendix (code)

## Sub 1