**Text

Description automatically generated**

**Compiler**

Wayne Borg (445103L) \*

\*B.Sc. (Hons) Software Development

Study-unit: **Compiler Theory and Practice**

Code: **CPS2000**

Lecturer: **Dr. Sandro Spina**

**FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We\*, the undersigned, declare that the assignment submitted is my / our\* work, except where acknowledged and referenced.

I / We\* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

\* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Course Code Title of work submitted

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date

# Introduction

This project aimed to build a compiler that translates an input source code file using the imperative language PixArLang, to the assembly-like language PixIR.

Diagram

Description automatically generated

1. The compilation process works as follows; the user provides a source code file.
2. A lexer and parser are initiated, the parser gets tokens from the lexer and produces an AST tree.
3. The AST tree goes through the following passes in order.
   1. XML generation
   2. Semantic analysis
   3. Code generation
4. The result of the code generation pass is saved to a file where it is accessible to the user.

# Frontend

The front end of the compiler starts by initializing a CharacterProvider to read this source file. This character provider uses Java’s RandomAccessFIle, which allows us to read a file using a pointer that can go forward or backward as needed. These operations are then used by the lexer as needed.

A lexer is also initialized, taking the CharacterProvider as its input. The lexer has the method nextToken which reads the next valid token from the characters returned by the character provider.

A parser is initialized taking the lexer as its input. The parser contains a parse method that produces an AST tree using the full input source.

## Lexer

The lexer is implemented using classification table and a table-driven DFA of the micro-syntax of the language, using the following DFA.

### Classification table

The characters are classified in the following classes as follows.

|  |  |
| --- | --- |
| character | class |
| 0 | Digit |
| 1 | Digit |
| 2 | Digit |
| 3 | Digit |
| 4 | Digit |
| 5 | Digit |
| 6 | Digit |
| 7 | Digit |
| 8 | Digit |
| 9 | Digit |
| A | AtoF |
| B | AtoF |
| C | AtoF |
| D | AtoF |
| E | AtoF |
| F | AtoF |
| a | AtoF |
| b | AtoF |
| c | AtoF |
| d | AtoF |
| e | AtoF |
| f | AtoF |
| G | GtoZ |
| H | GtoZ |
| I | GtoZ |
| J | GtoZ |
| K | GtoZ |
| L | GtoZ |
| M | GtoZ |
| N | GtoZ |
| O | GtoZ |
| P | GtoZ |
| Q | GtoZ |
| R | GtoZ |
| S | GtoZ |
| T | GtoZ |
| U | GtoZ |
| V | GtoZ |
| W | GtoZ |
| X | GtoZ |
| Y | GtoZ |
| Z | GtoZ |
| g | GtoZ |
| h | GtoZ |
| i | GtoZ |
| j | GtoZ |
| k | GtoZ |
| l | GtoZ |
| m | GtoZ |
| n | GtoZ |
| o | GtoZ |
| p | GtoZ |
| q | GtoZ |
| r | GtoZ |
| s | GtoZ |
| t | GtoZ |
| u | GtoZ |
| v | GtoZ |
| w | GtoZ |
| x | GtoZ |
| y | GtoZ |
| z | GtoZ |
| . | Point |
| # | Pound |
| \_ | Underscore |
| \* | Asterisk |
| / | Slash |
| + | Plus |
| - | Minus |
| > | GT |
| < | LT |
| = | Equals |
| ! | Exclamation |
| ( | BracOpen |
| ) | BracClose |
| : | Colon |
| ; | SemiColon |
| { | CurlyBracket  Open |
| } | CurlyBracket  Close |
| , | Comma |

### DSA

This is the DSA of the micro-syntax of the language

A picture containing drawing, sketch, diagram, pattern

Description automatically generated

### Implementation

The lexer uses 2 provided CSV files containing the classification table and the lexer transitions. These files where extracted to CSV files because they are easier to modify if any future versions the PixArLang requires any changes to the classifications and lexer transitions, without having to touch the code.

The lexer read characters from the character provider to form a lexeme. It reads characters, classifies them, and simulates the DSA until it reaches an error state, then rolls back to get the longest acceptable lexeme, such that the DSA ends in an accepted state.

The state that the DSA ends on determines the type of the token returned. For most accepted states this is simple, as there is a 1-1 relationship between the state and the token type. For some accepted states such as the ‘word’ and ‘sysfunc’ states, the type of the token is determined by the lexeme, depending on if the lexeme matches a keyword. If the DSA was never in an accepted state, then a “SyntaxErrorException” is returned to inform the user that the given lexeme couldn’t be understood by the lexer.

### Testing

Ensuring that all the tokens were implemented correctly was important as a mistake in the lexer could be hard to discover, therefore unit tests were implemented. A map of lexemes and their expected token type was created, and the Junit 5 test framework was used to test each lexeme.

2 tests where formed.

1. singleToken tests each token to ensure that the token type, start and end position, and the lexeme were correctly read.
2. singleTokenWithPadding tests each token, while adding a random amount of leading and trailing whitespace, to test the lexer’s handling of whitespace.

A MockCharProvider was created to serve as implementation of CharProvider, to be passed as input to the lexer. The MockCharProvider takes a string input and provides character inputs to the Lexer, as if it was reading the string from a regular file.