**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. The character provider also provides a LineNumberProvider which gives the ability to get the line, and column number from the character position. This is useful when showing errors to the user and is also used in the comments of the compiled PixIR program.

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.

### End of file

If more tokens are requested after the end of file has been reached, null is returned.

### Error handling

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. If the lexeme finishes in an accepted state, but the accepted state cannot map the lexeme to a valid tokenType (ex. the DSA finishes on ‘sysfunc’ state, but the lexeme is not a valid function ex. ‘\_\_abc’ ), 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. This was deemed feasible as, unlike other components of the compiler, there are a limited number of cases that need to be tested for the lexer.

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.

## Parser

The parser reads tokens from the lexer and uses them to form an abstract syntax tree.

### Parse Rules

The parser builds the AST tree using the following parse rules:

Note that in the RHS column, the red text indicates that the item refers to a token as described in the Lexer chapter, while the black text refers to other rules inside this table.

The ‘First’ Column indicates the types of tokens that should be expected first when parsing a specific rule.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| row | LHS | ::= | RHS |  | First |
|  |  |  |  |  |  |
| 1 | TypeLiteral | ::= | FloatType| IntegerType| BoolType| ColourType |  | FloatType| IntegerType| BoolType| ColourType |
| 2 |  |  |  |  |  |
| 3 | BooleanLiteral | ::= | True | False |  | True | False |
| 4 | IntegerLiteral | ::= | Int |  | Int |
| 5 | FloatLiteral | ::= | Float |  | Float |
| 6 | ColourLiteral | ::= | Colour |  | Colour |
| 7 | PadWidth | ::= | PadWidth |  | PadWidth |
| 8 | PadHeight | ::= | PadHeight |  | PadHeight |
| 9 | PadRead | ::= | PadRead Expr Comma Expr |  | PadRead |
| 10 | PadRandI | ::= | PadRandI Expr |  | PadRandI |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 | Literal | ::= | BooleanLiteral | IntegerLiteral | FloatLiteral | ColourLiteral | PadWidth | PadHeight | PadRead |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead |
| 14 | Identifier | ::= | Identifier |  | Identifier |
| 15 |  |  |  |  |  |
| 16 | Unary | ::= | (Subtract | Not) Expr |  | Subtract | Not |
| 17 |  |  |  |  |  |
| 18 | SubExpr | ::= | BracOpen Expr BracClose |  | BracOpen |
| 19 |  |  |  |  |  |
| 20 | ActualParams | ::= | Expr ActualParams\_ |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead| Identifier (function call or variable) |Subtract | Not |BracOpen | PadRandI |
| 21 | ActualParams\_ | ::= | (Comma expr ActualParams\_)|ε |  | Comma | ε |
| 22 | FunctionCall | ::= | Identifier BracOpen (ActualParams|ε) BracClose |  | Identifier |
| 23 |  |  |  |  |  |
| 24 | Factor | ::= | Literal | Identifier | FunctionCall | Unary | SubExpr | PadRandI |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead| Identifier (function call or variable) |Subtract | Not |BracOpen | PadRandI |
| 25 |  |  |  |  |  |
| 26 | Term | ::= | Factor Term\_ |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead| Identifier (function call or variable) |Subtract | Not |BracOpen | PadRandI |
| 27 | Term\_ | ::= | ((Multiply|Divide|And) Factor Term\_) | ε |  | Multiply|Divide|And |ε |
| 28 |  |  |  |  |  |
| 29 | SimpleExpr | ::= | Term SimpleExpr\_ |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead| Identifier (function call or variable) |Subtract | Not |BracOpen | PadRandI |
| 30 | SimpleExpr\_ | ::= | ((Add|Subtract|Or) SimpleExpr\_) | ε |  | Add|Subtract|Or|ε |
| 31 |  |  |  |  |  |
| 32 | Expr | ::= | SimpleExpr Expr\_ |  | True | False | Int | Float | Colour | PadWidth | PadHeight | PadRead| Identifier (function call or variable) |Subtract | Not |BracOpen | PadRandI |
| 33 | Expr\_ | ::= | ((NE|EQ|GT|GTE|LT|LTE) Expr\_) | ε |  | NE|EQ|GT|GTE|LT|LTE | ε |
| 34 |  |  |  |  |  |
| 35 | Assignment |  | Identifier Equals Exrp |  | Identifier |
| 36 | VarDecl | ::= | Let Identifier Colon TypeLiteral Equals Expr |  | Let |
| 37 |  |  |  |  |  |
| 38 | Print | ::= | Print Expr |  | Print |
| 39 | Delay | ::= | Delay Expr |  | Delay |
| 40 |  |  |  |  |  |
| 41 | PixelRange | ::= | PixelRange Expr Comma Expr Comma Expr Comma Expr Comma Expr |  | PixelRange |
| 42 | Pixel | ::= | Pixel Expr Comma Expr Comma Expr |  | Pixel |
| 43 |  |  |  |  |  |
| 44 | Return | ::= | Return Expr |  | Return |
| 45 |  |  |  |  |  |
| 46 | If | ::= | If BracOpen Expr BracClose Block (Else Block | ε) |  | If |
| 47 | For | ::= | For BracOpen (VarDecl| ε) SemiColon Expr SemiColon (Assignment | ε) BracClose Block |  | For |
| 48 | While | ::= | While BracOpen Expr BracClose Block |  | While |
| 49 |  |  |  |  |  |
| 50 | FormalParameter | ::= | Identifier Colon TypeLiteral |  | Identifier |
| 51 | FormalParams | ::= | FormalParameter FormalParams\_ |  | FormalParameter |
| 52 | FormalParams\_ | ::= | ((Comma) FormalParameter FormalParams\_) | ε |  |  |
| 53 |  |  |  |  |  |
| 54 | FunDecl | ::= | Fun Identifier BracOpen (FormalParams | ε) BracClose Arrow TypeLiteral Block |  | Fun |
| 55 |  |  |  |  |  |
| 56 | Statement | ::= | (VarDecl SemiColon) | (Assignment SemiColon) | (Print SemiColon) | (Delay SemiColon) |(Pixel SemiColon) | (PixelRange SemiColon) | If | For | While | (Return SemiColon) | FunDecl | Block |  | Let | Identifier | Print | Delay | PixelRange | Pixel | If | For | While | Return | Fun | CurlyBracOpen |
| 57 | StatementList | ::= | Statement StatementList\_ |  | Let | Identifier | Print | Delay | PixelRange | Pixel | If | For | While | Return | Fun | CurlyBracOpen |
| 58 | StatementList\_ | ::= | (Statement StatementList\_) | ε |  | Let | Identifier | Print | Delay | PixelRange | Pixel | If | For | While | Return | Fun | CurlyBracOpen |
| 59 |  |  |  |  |  |
| 60 | Block | ::= | CurlyBracOpen StatementList CurlyBracClose |  | CurlyBracOpen |
| 61 | Program | ::= | StatementList |  | StatementList |

### Implementation

The main class creates a parser and passes a lexer as input. A ‘ParseRule’ interface was used to provide a standard interface for each of these parse rules. Each parse rule listed above was implemented as its own class which extended the ‘ParseRule’ interface and had a ‘parse’ method which returned an AST node. In the ‘parse’ method, the parse rule class was responsible for consuming, looking ahead and calling other parse rules to generate the required AST node. In the event that the parse rule needed some parameters passed to it in order to work, these were passed in its constructor.

The parser creates a parser context which is passed to each parse rule. This parser context has the functionality to read tokens from the lexer using the ‘nextToken’ method of the lexer and use them for lookaheads and consuming tokens. The parser has a buffer of tokens that it uses for its lookahead. If the parse rule requests to consume a token, or to look a number of tokens ahead, and the buffer doesn’t have enough tokens to satisfy this request, then the buffer uses the ‘nextToken’ method to get more tokens from the lexer.

The parser context contains methods to skip comments when consuming or looking ahead. This allows the parser rule to avoid handling of comments, and therefore reduces complexity from the parse rules.

The parser starts its parsing by calling the parse method on the Program parse rule, which in turn calls any subsequent parse rules as required. This returns a Program AST node which is the root of the AST tree representing the program.

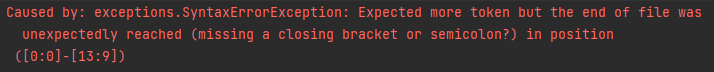
All the parse rules used can be categorized into 3 different classes.

1. The standard parse rule, which returns a specific AST node which reflects what the rule is, ex. The ‘While’ parse rule, consumes the tokens necessary for a while loop, and returns a ‘While’ AST node.
2. The ‘pass-thru’ parse rule, which uses lookaheads to determine which parse rule to call, and returns the AST node of that parse rule without encompassing it in its own. Ex. The ‘Factor’ parse rule, which can return a ‘Literal’, ‘SubExpr’, ‘Random’ etc.
3. The ‘chain’ parse rule, which uses 2 parse rules to parse chains, but ultimately returns a single ASTnode. These are used when the AST node requires an array such as ‘FormalParams’ parse rule, or when the rule required left-precedence such as the ‘Expr’ parse rule. In both cases, the main parse rule gets the first element and passes it to the inner parse rule, which has the same name of the main parse rule with an appended underscore. The inner parse rule keeps using the lookahead and recursively calls itself, until the lookahead determines that the end of the chain has been reached.

### Error handling

If there is a mismatch between a token that the parse rule is expecting, and the token returned by the parser, a ‘SyntaxErrorException’ is returned, giving the user details on the lexeme used to create the problematic token, and its location in the form of line number and column to help the user debug the error.



If the lexer returns a null token, indicating that the end of the file has been reached, and therefore it can give no more tokens, an error is returned to the user, hinting at the possibility of a missing closing bracket or semicolon, as they are the most common cause for this error. 

### Testing

Due to the infinite number of possibilities that the parser could encounter unit testing was deemed unfeasible. Therefore, testing of this component was mostly Ad Hoc, mostly by testing inputs which were deemed to have a high probability of being problematic.

# Intermediate language

The abstract syntax tree acts as an intermediate language between the frontend and the backend of the compiler. It uses a visitor design pattern using the ‘acceptVisitor’ function to allow different visitors to traverse the AST tree.

# Backend

The backend uses different visitors to iterate through the AST tree. First the ast is visited by a visitor which transforms it to XML, then by a visitor which performs semantic analysis to ensure that the AST tree is semantically correct. Finally the tree is traversed by a visitor which traverses the tree to generate the compiled code.

## XML visitor

The xml visitor uses a depth first traversal of the tree, which traverses each node in the array, to transform it into XML. The visitor also keeps a variable to keep the amount of indentation required.

When a node is reached, it uses a string builder to create the XML string representing the node.

The visitor first adds the opening tag of the node, then it increases the indentation variable, before calling the children, so that the children are indented a step further. After appending the result of the children to the string, the indentation is decremented to revert to the previous indentation value before the closing tag of the node is added.

Since some nodes such as the for loop and if statement nodes are allowed to have some null values, these are represented in the XML output as “<Null />”

The visit ultimately returns a string which can be used by the parent, or if the visit was performed on the root, the return value is the final XML representing the AST of the program.  
This XML is outputted to the console to help the user debug any issues with their code.