**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

Contents

[2. Introduction 5](#_Toc135321820)

[2.1. Using the compiler 5](#_Toc135321821)

[2.1.1. Requirements 5](#_Toc135321822)

[2.1.2. Setup 6](#_Toc135321823)

[2.1.3. Compiling and running 6](#_Toc135321824)

[3. Frontend 7](#_Toc135321825)

[3.1. Lexer 7](#_Toc135321826)

[3.1.1. Classification table 7](#_Toc135321827)

[3.1.2. DSA 1](#_Toc135321828)

[3.1.3. Tokens 2](#_Toc135321829)

[3.1.4. Implementation 2](#_Toc135321830)

[3.1.5. End of file 2](#_Toc135321831)

[3.1.6. Reserved words 2](#_Toc135321832)

[3.1.7. Error handling 2](#_Toc135321833)

[3.1.8. Testing 2](#_Toc135321834)

[3.2. Parser 2](#_Toc135321835)

[3.2.1. Parse Rules 2](#_Toc135321836)

[3.2.2. Implementation 5](#_Toc135321837)

[3.2.3. Error handling 7](#_Toc135321838)

[3.2.4. Testing 7](#_Toc135321839)

[4. Intermediate language 7](#_Toc135321840)

[4.1. AST nodes 8](#_Toc135321841)

[5. Backend 8](#_Toc135321842)

[5.1. XML visitor 8](#_Toc135321843)

[5.2. Semantic analysis 9](#_Toc135321844)

[5.2.1. Visitors 9](#_Toc135321845)

[5.2.2. Types 9](#_Toc135321846)

[5.2.3. Scopes 11](#_Toc135321847)

[5.2.4. Paths that return 13](#_Toc135321848)

[5.2.5. Semantic Errors 13](#_Toc135321849)

[5.2.6. limitation 15](#_Toc135321850)

[5.3. Code generation 15](#_Toc135321851)

[5.3.1. Visit result 15](#_Toc135321852)

[5.3.2. Memory 15](#_Toc135321853)

[5.3.3. Compiled Functions 16](#_Toc135321854)

[5.3.4. Traversing the AST tree 16](#_Toc135321855)

# 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 by placing the code inside ‘in.txt’.
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 called ‘output.txt’. where it is accessible to the user.
   1. The XML generation pass saves its result to a file called ‘intermediateXML.xml’

## Using the compiler

### Requirements

To use the compiler, please ensure that you have a JDK (<https://www.oracle.com/java/technologies/downloads/>) installed on your machine and that the ‘JAVA\_HOME’ environment variable is set (<https://www.baeldung.com/java-home-on-windows-7-8-10-mac-os-x-linux>).   
Please ensure that your JDK is at least JDK 8 or higher.

### Setup

To set up the compiler on your machine, please extract the source code into a separate folder.

### Compiling and running

To compile a PixarLang program into PixIr code, follow the following steps:

1. Copy the PixarLang code into the in.txt file. Please ensure that the file ends in a blank line. I.e. add a newline character as the last character of the file, not adding this character may result in a deadlock.
2. Open a command prompt and run the command ‘gradlew run’
3. The compiler will either compile successfully or return an error to indicate a problem with your code.
   1. If the compiler successfully compiles, successfully a success message is displayed 
   2. If an error was found within the input file, the error is displayed in the console. Ex:A picture containing text, font, software, screenshot

      Description automatically generated
4. The XML representation of the AST tree is saved to ‘intermediateXML.xml’ – This was done for Task 3
5. The PixIR code can be found in the ‘output.txt’ file.
6. To run the PixIR code, simply copy the code from output.txt to the VM.

# 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 backwards 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 a classification table and a table-driven DFA of the micro-syntax of the language, using the following DFA.

### Classification table

The characters are classified into 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

### Tokens

When a token is created, it is given a type to indicate what type of token it is, pointers to the start and end positions of the lexeme in the source code file which is used to create the error message in case of a syntax or semantic error, and the lexeme, which is the raw string that was parsed to form this token.

The following is a list of token types that the lexer maps a token to. The items in this list are implemented as an enum. When a token object is created, it is given one of these types.

**literals**

* Float
* Int
* Colour

**reserved words**

* True
* False
* Retrn
* If
* Else
* For
* While
* Fun
* FloatType
* IntegerType
* BoolType
* ColourType

**identifier**

* Identifier

**operations**

* Divide
* Multiply
* Add
* Subtract
* Arrow
* And
* Or
* Not
* Let

**relops**

* LT
* GT
* EQ
* NE
* LTE
* GTE

**punctuation**

* BracOpen
* BracClose
* CurlyBracOpen
* CurlyBracClose
* SemiColon
* Colon
* Comma
* Equals

**VM operations**

* PadHeight
* PadRandI
* PadRead
* PadWidth
* Print
* Delay
* PixelRange
* Pixel

**Comments**

* SingleLineComment
* MultiLineComment

### Implementation

The lexer uses 2 provided CSV files containing the classification table and the state transitions DSA. These files were extracted to CSV files because they are easier to modify if any future versions of the PixArLang require 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 acceptable state.

The state that the DSA ends on determines the type of token returned. For most accepted states this is simple, as there is a 1-1 relationship between the state and the token type. However, 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 the file has been reached, null is returned.

Note that in this implementation, a program has to have at least 1 statement. If the end of the file is reached, before a single statement, an error will return. Similarly happens with a block. A block needs at least 1 or more statements.

### Reserved words

The following is a list of words that are reserved as they have a specific meaning. These words cannot be used as variable or function names. This list is case insensitive, meaning that all variations of the capitalization of any word in this list are still not allowed.

* true
* false
* float
* int
* bool
* colour
* and
* or
* not
* let
* return
* fun
* if
* else
* for
* while

### 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 token type (ex. the DSA finishes on the ‘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.



A picture containing text, font, screenshot

Description automatically generated

### 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 were formed.

1. singleton tests each token to ensure that the token type, start and end position, and 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 an 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. Care was taken to ensure that none of these parse rules was left recursive:

Note that both the program and the block rules need at least 1 or more statements. A program or a block with 0 statements will return an error. This was done to avoid the common error of accidentally leaving a block empty.

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 parser is implemented as a top-down LL(k) parser. While most parse rules can be parsed using only 1 token of lookahead, some parse rules require 2 tokens. Theoretically, a parse rule can request to get as many tokens as needed to use as lookahead, as there are no restrictions on the number of symbols to lookahead.

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. If the parse rule needed some parameters passed to it 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 comments and therefore reduces the complexity of 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, 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, can return a ‘Literal’, ‘SubExpr’, ‘Random’ etc.
3. The ‘chain’ parse rule, 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 the ‘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 as 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.

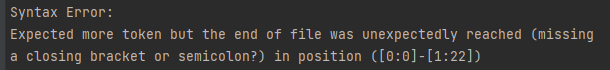
The parse rules create AST nodes, which are explained in Section 4

### 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.

A picture containing text, font, screenshot, typography

Description automatically generated

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 front end and the back end of the compiler. It uses a visitor design pattern using the ‘acceptVisitor’ function to allow different visitors to traverse the AST tree.

## AST nodes

Each type of node is implemented in its own class which extends ASTNode. When creating an AST node, any child nodes have to be passed ex. when creating an ‘if’ AST node, the conditional expression, then block, and else block have to be passed.

Additionally, the start and end locations of the source code have to be given, as they are used when an error needs to be returned to the user.

# 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, and then by a number of visitors 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, and saved to a file, to help the user debug any issues with their code.

## Semantic analysis

The semantic analysis is responsible for detecting errors related to types, variable scopes, function parameters, function returns etc. Any errors that manage to get past the semantic analysis pass will result in undetermined behaviour during the code generation pass and subsequently during runtime.

Whenever the semantic analysis finds an error, it is reported directly to the user, and the analysis stops without searching further.

### Visitors

The semantic analysis pass uses 3 visitors to help detect issues with the source code.

1. FunctionDeclarationVisitor – starts from the root of the AST node and traverses the tree looking for function declarations. When a function declaration is found it is stored in a hash map where the key is the identifier and the value is a FunctionDeclarationProperties object which stores the return type of the function, and the types of the expected parameters.  
   During this process, the FunctionDeclarationVisitor also reports a semantic error when 2 functions are declared using the same identifier.
2. SemanticVisitor – The main visitor, which starts from the root and traverses the tree looking for type errors, variable scope errors etc. When the visitor is created the map of functions that was generated by the FunctionDeclarationVisitor is passed as a parameter, as it is needed by the SemanticVisitor to find errors relating to function calls such as passing wrong parameters or expecting a different return type than the function returns.

When visiting a node, a node returns a visit result containing the type of its return value. In some rare occasions, the node can return an array of values instead of a single value.

1. ReturnVisitor – This visitor doesn’t start from the root, instead, a ReturnVisitor is created and used for every function declaration encountered. This visitor traverses the tree to analyse whether all paths lead to a return value. It finds errors where the function does not reach a return statement.

### Types

There are 4 variable types that are recognized by the compiler

* Bool
* Colour
* Int
* Float

The following table provides a comprehensive list of the types required as inputs to each type of node, and the type that the nodes are expected to output. If the return type of a child node doesn’t match the expected child type of its parent, an error is thrown.

This is checked when the parent is visited by the SemanticVisitor. After traversing a child, the parent compares the return type of the visit result, to the return type that was expected. In most cases, a float type is also able to accept an int type since it can be cast without any loss of precision.

The return type of a ‘FunctionCall’ AST node, is determined by the functions map provided by the FuncionDeclaration visitor.

The return type of an ‘Identifier’ node depends on the value stored in the symbol table.

When a function declaration node is visited, a variable is set to mark the expected value of any return nodes inside that particular function.

|  |  |  |
| --- | --- | --- |
| Node | Children types | Return Type |
|  |  |  |
| ActualParams | Same as Formal Params when the function was declared | N/A |
| Assignment | Depending on the symbol table | N/A |
| BinaryOp | Various | Float, Integer or Boolean |
| Block | N/A | N/A |
| BooleanLiteral | N/A | Boolean |
| ColourLiteral | N/A | Colour |
| Delay | Integer | N/A |
| Factor | Any | any – same as child |
| FloatLiteral | Float or Integer | Float |
| For | Expr has to be Boolean | N/A |
| FormalParameter | N/A | N/A |
| FormalParams | N/A | N/A |
| FunctionCall | N/A | Any – Depending on the type provided |
| FunDecl | N/A | N/A |
| Identifier | N/A | Any - depending on the symbol table |
| If | Expr has to be Boolean | N/A |
| IntegerLiteral | N/A | Integer |
| Literal | N/A | any – same as child |
| Negative | Float or Integer | Float or Integer – same as the input |
| Not | Boolean | Boolean |
| PadHeight | N/A | Integer |
| PadRandi | N/A | Integer |
| PadRead | Integer x2 | Colour |
| PadWidth | N/A | Integer |
| Pixel | Integer x 2, Colour | N/A |
| PixelRange | Integer x 4, Colour | N/A |
| Print | Any | N/A |
| Program | N/A | N/A |
| Return | Any – depending on the return type of the parent function | N/A |
| Statement | N/A | N/A |
| StatementList | N/A | N/A |
| SubExpr | Any | any – same as child |
| TypeLiteral | N/A | N/A |
| VarDecl | Depending on the symbol table | N/A |
| While | Expr has to be Boolean | N/A |

### Scopes

In this compiler, 2 types of memory management exist.

* The semantic visitor uses scopes, which are represented as a stack of scopes as the symbol table. Each scope is a place where there can two variables can never have the same identifier. These scopes are specially engineered to meet specific requirements such as the rule that a variable cannot be declared if the same identifier is already in use as an identifier of its function
* The code generator, and subsequently the VM during runtime uses frames. A frame is designed to be more practical such that additional frames can be opened for convenience, even if the variables in these frames are in the same scopes. That is to say that a scope maps to 1 or more frames.

The following table defines the tokens in our app which create a new frame/scope.

|  |  |  |
| --- | --- | --- |
| Token name | Frames/Scope | Notes |
| FunDecl | 1 single scope consisting of:   * 1 frame to hold params. * block frame | During compilation, a frame has to be added to the symbol table to hold the parameters.  During runtime, the params frame is created by the call instruction and does not need an ‘oframe’ command.  The block frame and any other unclosed frames have to be closed by the return function, as the ‘ret’ instruction only closes 1 frame (the param frame) – the block ast node, and any if, else, or loops surrounding the return statement will generate their own cframe instructions, but these will never be reached as the function will have returned.  During semantic analyses, the params scope has to be merged with the block scope. The frameReach gets reset so that we can only access the param scope, and any scopes inside the block |
| Block | 1 new frame/scope to hold variables inside the block |  |
| Program | 1 new frame/scope to hold variables |  |

The scope checking is handled by the SemanticVisitor, and to keep track of the scopes, a stack of lists is used. These lists store VarSymbol objects which store the type and identifier of a variable.

To ensure that no variables from outside the allowed scopes are used, a scopeReach variable keeps track of the number of scopes that the code has access to at any given point.

Different scopes can have different variables with the same identifier, in which case the variable in the topmost scope is used.

The blockMergeScopeFlag variable indicates to the block node, that it should not create a new scope, but keep adding to the topmost scope.

### Paths that return

The ReturnVisitor traverses the tree to analyse whether all paths lead to a return value. It finds errors where the function does not reach a return statement.

It starts from the block node of a function declaration.

Each node returns true if the node is guaranteed to reach a return statement.

While traversing the tree:

* A return node returns true since it is itself a return statement
* An if statement returns true if and only if both the then clause and the else clause return true when visited. If the if statement is missing an else clause, then false is returned.
* Loops such as for and while return false as it is possible that their condition is never true, and thus any return statements inside of them may never be reached.
* Statement lists return true, if at least one of their children returns true, which means that one of their statements is guaranteed to reach a return statement.

### Semantic Errors

The following is a comprehensive list of errors complete with sample error messages, a description of when they occur, and technical details as to how the specific error is detected.

|  |  |  |
| --- | --- | --- |
| Error | Example of error message | Technical details |
| Function already exists | Function already exists with identifier 'p' in position ([29:5]-[29:5]) | The FunctionDeclerationVisitor builds a map of identifiers and the parameter types and the return type of each function. If an identifier is already in use when another declaration is being analyzed, then a function with that identifier already exists |
| Variable already defined | Variable 'i' is already defined within this scope in position ([5:1]-[5:13]) | When the SemanticVisitor attempts to add the variable to the topmost scope, it finds another variable with the same identifier, which is not allowed. |
| Parameter already exists | Another parameter already exists with the name 'i' in position ([6:13]-[6:13]) | When the SemanticVisitor attempts to add the parameter to the topmost scope while visiting a function declaration node. It finds another parameter with the same identifier, which is not allowed. |
| The function is not defined | function 'doesNotExist' is not defined in position ([29:1]-[29:12]) | The semantic visitor tried to search for a function with the given identifier in the functions map, but it couldn’t find it. |
| Parameter count mismatch | Function 'f' takes '1' parameters, but was called with '2' parameters in position ([4:1]-[4:6]) | The function declaration specified that it requires a number of parameters and the types of those params were stored in the function map, but the params of the function call node had a different amount of children in the params |
| not all paths return a value | not all paths return a value in position ([10:1]-[15:1]) | After running the ReturnVisitor, we have determined that some paths do not reach a return statement. |
| Variable not defined | variable 'x' is not defined in this scope in position ([3:9]-[3:9]) | The identifier was not found in any scope in the symbol table. It either does not exist entirely, or it is currently inaccessible due to the scopeReach variable. |
| Return statement outside of a function | Return statement outside of function block in position ([43:1]-[43:8]) | A return statement was found when the currentFunctionReturnType was null, meaning that we were not inside a function declaration. |
| unknown type | unknown type in position ([9:7]-[9:17]) | The TypeLiteralASTNode’s value was null because the parser couldn’t map it to a known type |
| Type mismatch | Expected type '[Float, Int]' but got type 'Colour' in position ([9:15]-[9:21]) | The parent node was expecting a certain return type, however, when traversing the child, the visit result returned information that the return type of the child is different.   * For parameters inside a call function, the parameter types of the call node are matched with the parameters of the declaration. * For mathematical operators or system functions, the expected type is constant. * For return nodes, the expected value is that of the currentFunctionReturnType, which is set when visiting the function declaration AST node. * For variable assignments, the expected value is that which is stored in the symbol table.   Note that in most scenarios, an int is allowed when a float is expected, as it can be easily cast. |

### limitation

Most system functions such as pixel, pixelr, read, and delay expect integers. This means that a float can never be used with these functions as the target VM does not contain an instruction to round or truncate a float into an int. Therefore, given that dividing 2 integers returns a float, it is impossible to use division when preparing parameters from system functions.

Ex. It is impossible to fill half the screen using statements like

“\_\_pixelr 0,0,\_\_width,\_\_height/2,#000000”.

As height/2 would return a float and pixelr expects integers

## Code generation

The code generation pass traverses the AST tree, knowing that it is free from errors as the errors should have been caught by the semantic pass, and generates the required PixIR code.

### Visit result

When traversing the tree, each time a node is visited it returns a visit result, containing an array of strings, which are the instructions required to execute that specific node, in PixIR code.

### Memory

A memory table is kept in the form of a stack of lists, where the lists contain the identifiers of variables. These lists represent the frames that are available on the VM. Additionally, a function frameCountWhenFunctionDeclared keeps track of the number of frames that were presently opened when a function was declared. This is set at the beginning of a function declaration node, and unset at the end of the function declaration node. It is used in the return node, to help determine how many ‘cframe’ instructions are needed to clean up any unclosed frames before returning from the function. These unclosed frames are a result of the function block and any other blocks that the return function might be in such as conditional statements, or loops.

### Compiled Functions

When a function declaration node is visited, its code is stored in a list containing compiled functions, instead of returning it to its parent. These compiled functions are then appended to the end of the compiled program, by the visitProgramNode function.

### Traversing the AST tree

When traversing the tree, and visiting each node, there are 3 types of actions that the visit function can do

1. Temporary data

These types of nodes return temporary data which are not meaningful to the VM, but it was an easy way to get data to their parent nodes without adding additional infrastructure.

examples of these nodes are BooleanLiteralAstNode, ColourLiteralAstNode, FloatLiteralAstNode and IntegerLiteralAstNode, which simply pass their literal value to another node which then handles the transformation into actual executable instructions.

1. Pass through

These types of nodes are organizational nodes and simply pass through the instructions returned by their children.

Examples of these nodes are the SubExprAstNode, visitFactorAstNode and StatementAstNode

1. Fully functional code

The rest of the visit functions return a list of instructions which translates the node into fully executable instructions which perform mathematical operations, call VM functions such as print, width, pixel or read, or aggregate children such as the block node and the if, while and for nodes.

These nodes may be required to traverse their children before generating their code and integrating their children’s code within additional instructions.

* 1. Stack manipulation

Most instructions require some objects to be pushed or popped from the operands stack. Some nodes such as the identifier nodes, or the literal nodes, are themselves translated into push instructions to push items onto the stack. While other nodes such as the binary operation node, or the system function nodes translate into other instructions to manipulate the stack as required.

Each node makes sure to clean up after itself, making sure to not change values which do not belong to itself, as that could cause issues during the execution of other nodes. The only exception to this is the return node, which pushed the return value onto the stack, but the value may not be used, i.e. it is neither assigned to a variable nor used as a parameter to another operation. In this case, the return value stays at the bottom of the stack until the program terminates. Note that this is considered a memory leak, and therefore users of the PixarLang language are encouraged to consume the return values of functions.

* 1. Memory

Some nodes such as the block and program node require the use of ‘oframe’ and ‘cframe’ instructions, to open frames on the VM’s memory stack. These operations are reflected in the memory table of the CodeGenerationVisitor by adding and removing frames to the memory table as explained in Section 5.2.3. In most cases, the operations are in charge of closing their own frames, both in the VM’s memory and in the memory model, except for the return instruction which has to close any additional frames that were opened but not closed when we reach the return function, since the ’cframe’ instruction from any unclosed blocks will never be reached as explained in Section 5.3.2.

When a variable needs to be declared, it is added to the topmost frame.

When a variable is required to be assigned or accessed, the location of the variable is found by searching the memory table starting from the topmost frame. Therefore the memory table must be maintained to perfectly reflect the VM’s memory during runtime.

* 1. Jumping

Some nodes require jumps to be added to skip or repeat lines of code. Such as the if, while and for nodes. In these cases, the blocks to be jumped are created first, and then the jumps are created using the ‘#PC+\_’ notation taking into consideration the number of instructions of the generated blocks, to know how many instructions to jump.

* 1. Function calls

Function calls use the ‘.funcName’ notation to push the location of the function onto the stack.