Compiling `ParL` to `PArIR` in Rust

A report on the Rust compiler for the `ParL` programming language.

Giorgio Grigolo - 0418803L

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for making a false declaration may include, but are not limited to, 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).

Giorgio Grigolo
Student Name

Giorgio Grigolo
Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS2000
Course Code

Compiling ParL to PArIR in Rust
Title of work submitted

30/05/2024
Date

Contents

Abstract

In this report, we discuss the implementation details of a compiler for `ParL`, an expression-based strongly typed programming langauge. Code written in `ParL` is compiled to `PArIR`, which is the proprietary assembly-like language that is used to drive the programmable pixel art displays designed by the company `PArDis`. The `ParL` compiler was written in Rust, due to its strong type system and performance characteristics. This assignment was part of the course CPS2000 - Compiler Theory and Practcie and was submitted in partial fulfilment of the requirements for the degree of B.Sc (Hons.) in Computer Science and Mathematics at the University of Malta.

1 Project Structure

The ParL compiler is implemented in Rust, and is structured as a Rust crate providing a CLI interface. The folder structure of the project reflects the fact that the compiler was built incrementally, with each stage having the core implementation in its own module (folder).

1.1 Dependencies

The use of external crate dependencies has been kept to a minimum throughout this assignment. Currently, the only external dependencies are:

- clap: A crate that provides a convenient way to define command-line interfaces.
- thiserror: A crate that provides a convenient way to define custom error types.
- console: A crate that allows for coloured printing to stdout, for decoration purposes.

1.2 Modules

The ParL compiler is structured as a Rust crate, with the following modules:

- core: Contains the core data structures, such as the Token struct, the TokenKind enum and the AstNode enum, which are used throughout the compiler, in the majority of the stages.
- lexing: Contains the lexer implementation, together with its requirements such as the Dfsa and the Lexer struct.
- parsing: Contains the parser implementation, together with the Parser struct and the TreePrinter visitor.
- semantics: Contains a visitors folder, which in turn contains a number of visitors, namely the SemanticAnalyser, the Formatter, and the TreePrinter.
- generation: Contains the ParIR code generator visitor, as well as an abstraction of the ParIR language, instructions.

1.3 Quick Start

The `ParL` compiler has a command-line tool interface, and these are the commands that can be used to interact with. The initial run for any of these commands will cause the Rust project to be built, so the first run might take a bit longer than usual.

Getting help:

```
cargo run -- --help
```

Formatting a `ParL` file:

```
cargo run fmt path/to/file.parl
```

The above command will format the input file according to the a predetermined style guide, and directly modify the file in place.

Running the lexer:

```
cargo run lex path/to/file.parl
```

The above command will print to `stdout` the tokens that the lexer has extracted from the input file, one per line.

Running the parser:

```
cargo run parse path/to/file.parl
```

The above command will print to `stdout` the abstract syntax tree (AST) that the parser has generated from the input file, in a pretty-printed format provided by the `TreePrinter` visitor.

Running the semantic analyser:

```
cargo run sem path/to/file.parl
```

The above command will print to `stdout` any semantic errors and/or warnings that the semantic analyser has found in the input file.

Running the full compiler:

```
cargo run compile path/to/file.parl
```

The above command will compile the input file to `PArIR` and print the generated code to `stdout`. If the `-o output_file.parir` flag is provided, the output will be written to the specified file.

Note that all compilation related commands cause the previous steps to be run as well, although only the output of the requested step is printed to `stdout`.

2 Lexical Analysis

The first step in the compilation process is the lexical analysis. A recurring theme in the implementation of this compiler is the use of *abstraction* to achieve *modularity*, and simplify the implementation of the subsequent stages. The lexical analysis is no exception to this rule.

At the highest level, the lexical analysis is implemented as a `Lexer` struct, which is responsible for reading the input source code, and producing a stream (*or vector*) of tokens.

We shall start by defining the `Token` struct, which is a wrapper for the different types of tokens that can be found in a `ParL` program. The `Token` enum is defined as follows:

```
pub struct Token {
    pub kind: TokenKind,
    pub span: TextSpan,
}
```

The `TextSpan` struct is mainly used to store the lexeme of the token, and its position in the source code, which is useful for error reporting. The `TokenKind` is just an enum, which represents the different types of tokens, such as `Identifier`, `IntegerLiteral`, `If` and many more.

The actual `Lexer` struct is defined as follows:

```
pub struct Lexer<B: Stream> {
    buffer: B,
    dfsa: Dfsa,
}
```

Listing 1: The `Lexer` struct.

The `Lexer` struct is generic over the type `B`, which is a trait that abstracts out the file reading logic from the lexer. This is useful for testing purposes, as we can easily mock the file reading logic, and test the lexer in isolation. It is also useful in case the lexer is to be improved in the future, by using more efficient file reading logic, such as the double buffering technique.

The main function of the `Lexer` struct is the `next_token` function, which is responsible for reading the next token from the input stream. Not much can be said, as it has been heavily inspired by the `next_token` function in the textbook *Engineering a Compiler* [3]. We provide a wrapper around the `next_token` function, called `lex()`, which filters out comments, whitespace and newlines, and only returns the actual tokens, and if any, a list of errors that have been encountered.

We note that the `next_token` returns `Result<Token, LexicalError>` instead of just a `Token`, which is caught by the wrapper `lex()`. This feature, known as *synchronization*, allows the lexer to return an error, instead of crashing. This enables the lexer to recover after an error has been encountered, such as reading a character that is not in the character table of the DFSA. It will continue reading from the next token, and find other invalid characters.

Whilst using a compiler, one would expect to be given as many errors as possible, instead of playing whack-a-mole with each error one at a time.

2.1 DFSA Builder

The `Lexer` that was implemented in the project, was required to be a based on a finite state automaton, with a character table. Initially, the DFSA was implemented manually, but as the number of tokens (and hence states) gre, it was getting harder to keep track of the transitions.

To solve this problem, together with the `Dfsa` struct, a `DfsaBuilder` struct was implemented, which is responsible for dynamically building the DFSA from a number of chained function calls. I will describe each method of the `DfsaBuilder` struct, and how it is used, by demonstrating the effect as it is run incrementally on an empty DFSA.

The `DfsaBuilder` struct is defined as follows:

```rust
pub struct DfsaBuilder {
    pub max_state: i32,
    pub accepted_states: Vec<i32>,
    pub character_table: HashMap<char, Category>,
    pub transition_table: HashMap<(i32, Category), i32>,
    pub state_to_token: HashMap<i32, TokenKind>,
}
```

Clearly, we can see that states are represented by `i32` integers, and transitions are represented by a tuple of the current state and the category of the character that is being read. The category of a character is defined by the `Category` enum, containing variants such as `Digit`, `Letter`, `Whitespace`, and many more.

The first function is

```
add_category(&mut self, range: Vec<char>, category: Category)
```

This method is used to add multiple characters to the same category. For example, we can add all the digits to the `Digit` category, by calling `add_category('0'..='9', Category::Digit)`. Note that in the actual implementation, the `range` parameter is actually a more complex type, but for simplicity's sake it can be thought of a vector. It actually accepts anything that can be casted to an iterator, as to allow for more convenient usage. This is done using Rust's built in ranges like `'0'..='9'`. As it does not have any effect on the DFSA, but only modifies the character table, a DFSA will not be shown.

Then, we have

```
add_transition(&mut self, state: i32, category: Category, next_state: i32)
```

This method is used to add a transition from one state to another, given a category. For example, we can add a transition from state 0 to state 1, when reading a digit, by calling `add_transition(0, Category::Digit, 1)`. Note that it doesn't automatically make the state 1 an accepted state, as it is only responsible for adding transitions.

start
0
start
0
Digit
1

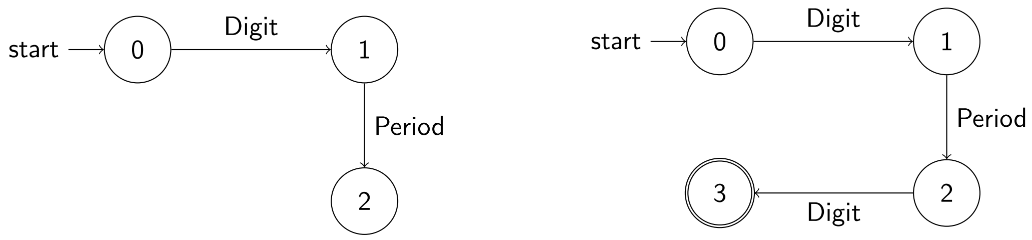
(a) The DFSA before adding calling `add_transition` (b) The DFSA after adding calling `add_transition`

The next function is

```
auto_add_transition(&mut self, state: i32, category: Category
          to: Option<i32>, token_kind: TokenKind) -> i32
```

This method is used used to add a transition from `(state, category)` to the state `to`. Since `to` is an `Option`, it can be `None`, in which case the destination state is automatically set to the next state. This is useful for adding transitions to the same state, as well as constructing slightly more complicated sequences of transitions, say for tokenizing a float. The `token_kind` parameter is used to set the token kind of the state, if it is an accepted state. Similarly to the `to` parameter, it can be `None`, in which case the destination state is not an accepted state. We also note that this function returns an integer, which is the state that was just added, for further use in subsequent calls.

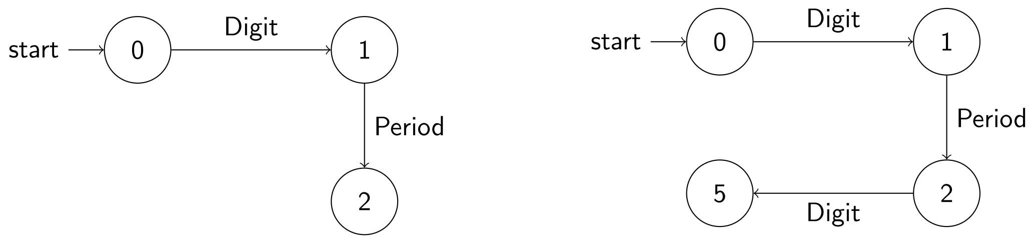
If we run `auto_add_transition(2, Category::Digit, None, Category::Float)`, the resultant DFSA will be as follows:



(a) Before running the function

(b) After running the function

If we run `auto_add_transition(2, Category::Digit, Some(5), None)`, the resultant DFSA will be as follows, meaning we just want to map to an auxiliary state that doesn't correspond to any token kind, (and thus is not an accepted state), the following is the resulting DFSA.

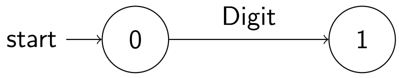


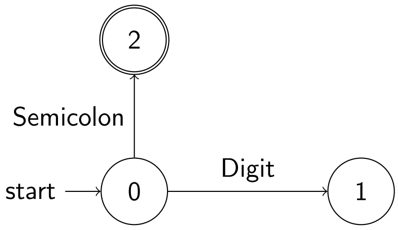
Next up, we have the function

```
add_final_character_symbol(character: char, category: Category, token_kind:
                    TokenKind)
```

This function is a shortcut for adding single characters that are final symbols, such as the semicolon, or the period to the character table, and the transition that follows from this to the transitions table.

Suppose we called `add_final_character_symbol(';', Category::Semicolon, TokenKind::Semicolon)`. The resultant DFSA would be as follows:





Then we have a function that does the same thing as the previous one, but is able to take in multiple tuples of the type (char, Category, TokenKind), and add them all at once. This function is called add_final_character_symbols.

Finally, we have the function build, which is used to build the DFSA from the builder. This function is called at the end of the chain of function calls, and returns the DFSA that was built.

Clearly, the DfsaBuilder has proved itself useful for an efficient and readable way to build the DFSA for the lexer. However, we still require a greater degree of freedom when it comes to making complex multi-character transitions.

> In the production rules for the ParL grammar, there is a slight inconsistency. There is an overlap between the Letter and the Hex symbol. This was taken into account by creating an auxiliary category called HexAndLetter, so that the lexer can differentiate between the two, on later stages in the transitions.

To be able to get around the above inconsistency, as well as to allow for the aforementioned flexibility, we can also call the .transition() method on the DfsaBuilder, and have it return an instance of the Transition struct. This transition automatically starts from the first unused state index, (which is kept track of in the max_state member of the DfsaBuilder struct) and gives us another set of functions we can chain to further build the transition.

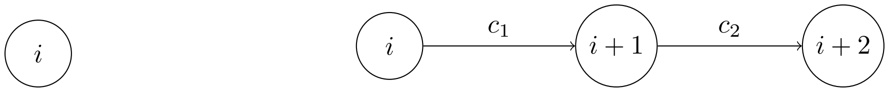
The Transition struct holds a reference to the DfsaBuilder that it was created from, the current state, the list of transitions that have been added, and the a list of final states with the token kind that they return.

```
pub struct Transition<'a> {
    dfsa_builder: &'a mut DfsaBuilder,
    current_state: i32,
    transitions: Vec<((Category, i32), i32)>,
    final_state_token: Vec<(i32, TokenKind)>,
}
```

The first function that can be called on the Transition struct is the following:

```
to(category: Category) -> Transition
```

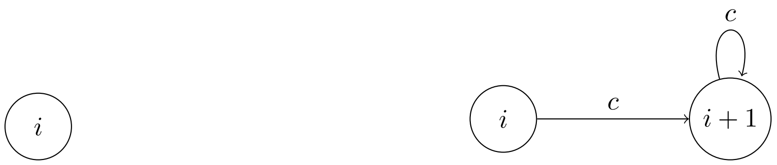
This method is used to just append a single category to the transition. Calling it multiple times, increments an internal pointer, that always points to the last state that was added. Thus, calling it multiple times creates a *chain* of states.



(a) The transition before adding calling `to`

(b) The transition after adding calling `to` twice with c_1 and c_2 respectively

Then, we have the `repeated()` method, which is used to allow an indeterminate number of repetitions of the last category, think of this as the `*` symbol in regular expressions.



(a) The transition before adding calling `repeated`

(b) The transition after adding calling `to(c)` followed by `repeated()`

Whenever we want to make the last state an accepted state, we can thenc all the

```
goes_to(token_kind: TokenKind)
```

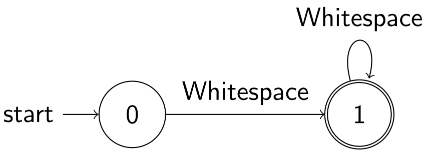
method, that simply marks the last state as an accepted state, and sets the token kind it returns to the one that was passed as an argument.

Finally, we have the `done()` method, which is used to append the transition that has been prepared to the actual DFSA from which `.transition()` was called from.

Full example

```
let mut builder = DfsaBuilder::new();
builder.add_category([' '], ['\t'], Category::Whitespace)
       .transition()
       .to([Category::Whitespace])
       .repeated()
       .goes_to(TokenKind::Whitespace)
       .done();
```

Would result in the following DFSA, with the category table already filled in with the whitespace characters.



The actual definitions of the DFSA that was used in the lexer can be found in `lexer.rs`, as part of the `Lexer::new()` function, and also in Appendix A. The final thing I must mention is how keyword are interpreted. Keywords are treated as identifiers, and then checked against a `HashMap` of reserved keywords. If the identifier is found in the map, the token kind is changed to the corresponding keyword.

3 Parsing the Token Stream

Parsing the token stream generated by the lexer is where the Rust language really shines. It allows to define an enum, called `AstNode`, which represents the different types of nodes that can be found in the abstract syntax tree of a `ParL` program. Additionally, we can mix and match different types of variants, such as ones with no data (similar to unit-like structs, like `struct Foo;`), ones with unnamed data (similar to tuple structs, like `struct Foo(i32, i32);`), and ones with named data (similar to regular structs, like `struct Foo { x: i32, y: i32 };`).

This gives us much more flexibility when defining the AST, in contrast to C or C++, where enums are basically just integers. Furthermore, I had to create a type alias from `Box<AstNode>` to `AstNodePtr`. This is because certain `AstNode` variants, have another `AstNode` as a child. The Rust compiler complains about the enum having infinite size, as it is can't be calculated at compile time. This is understandable, as for example, if the `AstNode::Expression` variant contained another `AstNode` as its member, the compiler would try to recursively visit this member to check its size, which could also be an expression, visiting the same member again, and so on. With a `Box`, the size of the enum is known at compile time, as it is just a pointer to the heap, and the compiler is happy. Lists of `AstNodes`, like parameter lists, do not need this, as a Rust `Vec` is already allocated on the heap.

The actual parser is implemented as a struct called `Parser`, which must be initialized with the list of tokens generated by the lexer. The parser privately defines all the different parsing functions. The only public function is `parse`, which is called to start the parsing process. The parser uses a recursive descent parsing algorithm, together with the Rust pattern matching feature, which is used to differentiate between the different variants of the `AstNode` enum.

Due to this, the Visitor design pattern is slightly different in Rust. With pattern matching, we avoid having every variant of the `AstNode` enum requiring it's own `accept` method. Instead we define each case in the `match` clause in the `visit` method for a given struct implementing the `Visitor` trait. Pattern matching is thoroughly used to ensure correctness and to avoid unnecessary code duplication.

A number of utility functions, such as `consume_if(kind:Tokenind)`, and `assert_token_is_any (kind: Vec<TokenKind>)`, are used to simplify the parsing process. The former consumes a token if it matches the given `TokenKind`, and returns an error otherwise. This is useful as it in allows the user to quickly identify syntax errors in the input program, with the parser suggesting what it was expecting, and announcing what it found instead. The latter function, asserts that the current token is any of the given `TokenKinds`, and returns an error if it is not. This is useful when the parser is expecting one of a number of different tokens, such as when declaring a variable, we expect either an `int`, `float`, `colour` or `bool` token kind after the semicolon. The error outputted will be more informative, as it will list all the possible token kinds that were expected.

The full list of the `AstNode` enum variants can be found in Appendix B.

3.1 Syntax Modifications

It is worth noting that the original grammar for the parser was not perfect. The precedence of the operators was not correctly defined, leading to incorrect parsing of expressions. For example, the expression

```
a < b and c >= d
```

was being parsed as

```
a < (b and c) >= d
```

which is expected from a programming language that treats the ‘and’ and ‘or’ keywords as its bitwise operators. After observing the behavior of these operators in the VM (they are indeed logical operators), the correct parsing should be

```
(a < b) and (c >= d)
```

The ‘and’ and ‘or’ operators were being parsed with a higher precedence than the comparison operators. Whilst researching alternative expression grammars, I stumbled upon the Lox language [1], and decided to take inspiration from it use its production rules for expressions.

The new production rules for expressions are as follows. All other rules remain the same.

$$
\begin{aligned}
\langle \text{Expression} \rangle &::= \langle \text{Equality} \rangle \ \{ \ (\ \text{‘and’} \mid \text{‘or’} \ ) \ \langle \text{Equality} \rangle \ \} \\
\langle \text{Equality} \rangle &::= \langle \text{Comparison} \rangle \ \{ \ (\ \text{‘!=’} \mid \text{‘==’} \ ) \ \langle \text{Comparison} \rangle \ \} \\
\langle \text{Comparison} \rangle &::= \langle \text{Term} \rangle \ \{ \ (\ \text{‘<=’} \mid \text{‘<’} \mid \text{‘>=’} \mid \text{‘>’} \ ) \ \langle \text{Term} \rangle \ \} \\
\langle \text{Term} \rangle &::= \langle \text{Factor} \rangle \ \{ \ (\ \text{‘+’} \mid \text{‘-’} \ ) \ \langle \text{Factor} \rangle \ \} \\
\langle \text{Factor} \rangle &::= \langle \text{Unary} \rangle \ \{ \ (\ \text{‘*’} \mid \text{‘/’} \mid \text{‘\%’} \ ) \ \langle \text{Unary} \rangle \ \} \\
\langle \text{Unary} \rangle &::= (\ \text{‘!’} \mid \text{‘-’} \ ) \ \langle \text{Unary} \rangle \mid \langle \text{Primary} \rangle \\
\langle \text{Primary} \rangle &::= \langle \text{Literal} \rangle \mid \langle \text{Identifier} \rangle \mid \langle \text{FunctionCall} \rangle \mid \langle \text{SubExpr} \rangle \\
&\quad \mid \langle \text{Unary} \rangle \mid \langle \text{PadRandI} \rangle \mid \langle \text{PadWidth} \rangle \mid \langle \text{PadHeight} \rangle \mid \langle \text{PadRead} \rangle
\end{aligned}
$$

One may also notice the addition of the % operator, known as the modulo. Although this wasn't specified in the original grammar, it is present as an instruction in the VM, so I decided to include it. In the context of drawing pixels, it's useful for creating a wrap-around effect in loops.

Additionally, it was also noted that in the VM's source code, the inc and dec instructions were present, although undocumented. These could've been well coupled with the ++ and – operators, also absent in the grammar. Alternatively, the following operators could've been defined, += and -=, for more Pythonic code. Although unimplemented, various ideas came to mind for the latter two operations, such as looping x times over the inc and dec instructions, where x is the evaluated expression on the right hand side of the operator. This allows for more compiler optimizations when generating the ParIR code.

4 Semantic Analysis

Semantic analysis is the process of checking the program for semantic errors, such as type errors, and ensuring that the program is well-formed. This is done after the parsing stage, where the abstract syntax tree has been constructed. The semantic analysis stage is where the compiler ensures that the program is semantically correct, and can be translated to the target language. This is done by traversing the abstract syntax tree, and checking that the program adheres to the rules of the language.

4.1 Visitor Trait

The `Visitor` trait has been mentioned in the previous section, but it is now crucial to define it and describe it properly.

```
pub trait Visitor<T> {
    fn visit(&mut self, node: &AstNode) -> T;
}
```

The `Visitor` trait is a generic trait that takes a type parameter `T` and defines a single method, `visit`, which takes a mutable reference to the struct that implements the trait, and a reference to an AST node. The method returns a value of type `T`.

The generic type parameter `T` is central to the way we will be using, or rather, re-using the visitor trait in the code generation stage. It will allow us to control the return type of the `visit` method, and thus, the type of the value that is returned when visiting a node in the abstract syntax tree. In other words, we can access, or evaluate, information about an AST node, depending on the struct that implements the `Visitor` trait.

I have implemented the `Visitor` trait on a number of structs (here on referred to as Visitors), and these are:

- `Formatter`: A visitor that parses, and then *un-parses* the AST, rewriting the source in an (un-configurable) opinionated style. Similar to running `cargo fmt`.
- `TreePrinter`: A visitor mainly used for debugging the parsing stage, which prints the AST in a vertical tree-like structure.
- `SemanticAnalyser`: The visitor that performs the semantic analysis on the AST, checking for type errors, and ensuring the program is well-formed.
- `PArIRWriter`: The visitor that generates the PArIR code from the AST.

In this report I will describe the latter two, as they are part of the requirements for the project.

4.2 Type/Scope Checking

The SemanticAnalyser, other than being a visitor, also contains a number of structs inside of it, which are used to keep track of the current scope, and the types of variables that are in scope.

```
struct SemanticAnalyser {
    symbol_table: Vec<SymbolTable>,
    inside_function: bool,
    scope_peek_limit: usize,
    results: SemanticResult,
}
```

The SemanticAnalyser struct contains a vector of SymbolTables, which functions as a stack. Each SymbolTable contains a mapping of variable names (lexeme) to their types (SymbolType). My implementation of SymbolType also further differentiates between identifiers that belong to a function, array, or primitive type variable definitions. In the case of functions, I stored the entire signature, including the return type, and the types of the parameters.

An implementation detail of the SymbolTable is that it is a sorted LinkedList, which allows for faster lookups. We ensure the list is ordered at the moment of insertion, where we lexicographically compare the lexemes of the identifiers we are storing in the symbol table.

Additionally, the SemanticAnalyser struct keeps track of whether the visitor is currently is inside a function, through the inside_function flag. Different behaviour is expected from the visitor when inside a function, for example, when checking for the existence of an identifier, we are only allowed to look for it in the current scope, and not in the global scope.

The scope_peek_limit field is used to limit the number of scopes that the visitor can look up when checking for the existence of an identifier. Used in conjunction with the inside_function flag, it allows the visitor to only look up to a certain number of scopes when inside a function, and all the way up to the global scope when outside of a function.

Finally, we have the results field, of the type SemanticResult. This is simply a struct that contains two lists of possible errors that can be emitted during the semantic analysis stage. We categorize these errors as Warnings, i.e. possible involuntary mistakes performed by the user that do not necessarily disallow the compilation of the program, and Errors, i.e. mistakes that must be fixed before the program can be compiled.

Variable Shadowing

This implementation of the ParL compiler allows for *variable shadowing*, in a similar fashion to Rust [2]. This means that a variable can be re-declared in an inner scope, and the latter will *take over* the former for the life-time of the inner scope. Variable shadowing will produce a warning, as it can lead to confusion, but variable re-declaration in the same scope will always produce an error.

Now that we know what tools the semantic analyser has at its disposal, we still have one more thing to discuss. How do we find out the types of arbitrary expressions, possibly with a mixture of literals, variables, function calls, and operations applied to them? To answer this question we must define the type inference of operations between types in the language. These are the types available.

`Int`

`Float`

`Colour`

`Bool`

`Array<T>`

Void

+ - * /	Int	Float	Colour	Boolean
Int	Int	Float	N/A	N/A
Float	Float	Float	N/A	N/A
Colour	N/A	N/A	Colour	N/A
Bool	N/A	N/A	N/A	N/A

(a) Addition, subtraction, multiplication, and division operations between types.

== != < > <= >=	Int	Float	Colour	Boolean
Int	Bool	Bool	N/A	N/A
Float	Bool	Bool	N/A	N/A
Colour	N/A	N/A	Bool	N/A
Bool	N/A	N/A	N/A	Bool

(b) Comparison operations between types.

%	Int	Float	Colour	Boolean
Int	Int	N/A	N/A	N/A
Float	N/A	N/A	N/A	N/A
Colour	N/A	N/A	N/A	N/A
Bool	N/A	N/A	N/A	N/A

(a) Modulo operation between types.

and or	Int	Float	Colour	Boolean
Int	N/A	N/A	N/A	N/A
Float	N/A	N/A	N/A	N/A
Colour	N/A	N/A	N/A	N/A
Bool	N/A	N/A	N/A	Bool

(b) Logical operations between types.

Every time we encounter an operation between two types, we must thus check the tables above to see if the operation is valid. If it is, we can infer the type of the operation, and continue checking the rest of the expression. If it is not, the semantic analyser must emit an error, and stop the compilation process.

Type Inference

This type inference was only possible due to the flexibility between the operations performed in the VM itself. Given the VM is implemented in JavaScript, this flexibility is warranted, as JavaScript is a dynamically typed language. Had it been done on actual hardware, the type inference would have been much more rigid, and the checking must have been more strict, possibly with more intricate conversions to achieve the desired result.

Behind the scenes, the compiler actually assigns a type to unsupported operations, namely the `Unknown` type, which by default is incompatible with any operation.

This is the general idea behind evaluating types of expressions in the `SemanticAnalyser` visitor. How do we determine however the true return type of a function declaration, or a `Block`? Well, we just traverse the `AST` until we hit a `Return` statement, and then we check the type of the expression that is being returned. Since we are not allowed to have functions that do not return anything, any `Block` is at first assigned the `Void` type, and if it remains so until the end of the block, an error is emitted.

With all the type information of the symbols present in the program acquired, the semantic analyser will check this against the type provided by the user in the function signature, or variable declaration, and will emit a type mismatch error, stating what the type it expected is (the one the user defined) and what it found, the type of the returned expression. In the case of arrays, lengths must also match.

Additionally, in the case of `For` loops, there is one additinoal thing to pay attention to. Even though the body of for loop is a block in its own right, we must actually start the scope of the for loop before the body, but then, we cannot visit the body with the normal rules. In fact, a new function called `visit_unscoped_block` was created to override the normal behaviour of visiting a block, and instead, visit it without creating a new scope. This is crucial for the for loop, as the variables declared in the for loop must be accessible in the body of the loop (and because we can't shadow the iterator variable).

Lastly, type-casting is also supported in the language, and is done by using the `as` keyword. The type-casting operation is only allowed between specific types, otherwise an error is emitted.

to from	Int	Float	Colour	Boolean
Int	✓	✓	✓	×
Float	×	✓	×	×
Colour	✓	✓	✓	×
Bool	✓	✓	×	✓

Table 3: Type-casting operations between types.

Limitation

The type-casting operation is only supported between the types listed in the table above. However, this may cause some overflows, which are not check for in the compiler. A clear example is taking the `Colour #ffffff`, casting it to an `Int`, adding one, and casting it back to a `Colour`. Drawing this on the display will result in a black pixel, as the `Int` overflowed.

A further improvement could be an overflow check implemented on types with arbitrary values that have fixed ranges, such as `Int` and `Floats`. In fact, whilst converting array indices from a lexeme to a Rust `usize`, we assume that the lexeme represents a valid `usize`. If it is too big, the compiler will crash due to an `unwrap()`.

To summarize, the following are the rules we shall impose on the language:

- Atomic variables, functions and arrays must be declared before they are used.
- Atomic variables, functions and arrays cannot be re-declared in the same scope.
- Atomic variable and array declaratinos can be shadowed in inner scopes.
- When declaring arrays, the number of elements must match the length of the array, so overallocation is not allowed.
- Functions cannot be called with the wrong number of arguments.
- Function parameters must match the types in the function signature.
- Functions must return a value.
- Functions must return the correct type.
- Operations between types must be valid.
- The type of the expression in a return statement must match the return type of the function.

5 Code Generation

The final step in the compilation process is the code generation phase. In this phase, the compiler takes the AST and generates the PArIR code that will be executed by the PArDis displays.

5.1 Abstracting Instructions

The first step in the code generation phase was to abstract the PArIR instructions into a Rust enum. This enum, called Instruction, contains all the instructions that the PArDis displays can execute. Some of the instructions, like Push, Pusha and really most of the push variants, take some arguments, which are stored in the enum itself, as an unnamed tuple e.g. PushValue(usize).

The Instruction enum has a Display trait implementation, which is equivalent to a toString method in other languages. We will set the string representation of the instruction to be the same as the PArIR instruction itself, with the arguments formatted accordingly, as it will be outputted in the final PArIR code.

5.2 Program Intermediate Representation

Fundamentally, we can represent a PArIR program as an ordered list of instructions. Furthermore, these instructions can be grouped in two sections:

- The *function section*, which contains all the function definitions.
- The *main section*, the entrypoint of the program, which contains the instructions that will be executed when the program starts.

In fact, our Program struct, is exactly the above.

```
struct Program {
    pub functions: Vec<Instruction>,
    pub main: Vec<Instruction>,
}
```

When writing the resulting PArIR code to a file, we first write the function section, followed by the main section. Due to the fact that the only thing we need to call a function, is its name, not the line number where it has been defined, we can successfully divide the program into these two sections.

5.3 Code Generation Visitor

The code generation visitor is the final visitor that is run on the AST. Contrary to the the semantic analyser, instead of returning type information as it visits the nodes of the AST, it returns *line numbers* of the generated PArIR code. These line numbers are mostly used to generate all kinds Jump instructions correctly. For example, when we generate an if statement, we need to know the line number of the first instruction of the true branch, and that of the false branch, so that upon evaluating the condition, we know where to jump.

Of course, other things need to be kept track of, like the current stack level, and the current frame index, which are used to generate the correct Push instructions when referencing variables from the symbol table. Here is the full definition of the PArIRWriter struct, upon which we implement the visitor trait.

```
struct PArIRWriter {
    /// Stack of symbol tables, each representing a scope
    symbol_table: Vec<SymbolTable>,
    /// The ParIR program container
    program: Program,
    /// Pointer to the current instruction
    instr_ptr: usize,
    /// The current stack level
    stack_level: usize,
    /// The current frame index
    frame_index: usize,
}
```

Additionally, we also have some helper functions such as

- get_scope_var_count, which returns the number of slots in the that we need to allocate in the stack frame for the current scope.
- push_scope() and pop_scope(), being common operations, we shall automate them, whilst resetting the frame index and incrementing the stack level.
- get_memory_location(token: Token), which returns the memory location of a variable, given its name, explained in the next section.

5.4 Memory Access

Even though we keep track of how many frames have been created in the stack_level field, the most recent one is always at index 0. Thus, we must calculate the relative memory location of a variable by subtracting the stack level of the variable from the current stack level. This is the corresponding function.

```
fun get_memory_location(&self, symbol: &Token) -> Option<MemoryLocation> {
    self.find_symbol(symbol).and_then(|s| {
        let relative_mem_loc = s.memory_location;
        relative_mem_loc.map(|mem_loc| MemoryLocation {
            stack_level: self.stack_level - mem_loc.stack_level,
            frame_index: mem_loc.frame_index }) }) }
```

6 Arrays

Implementing arrays was definetely a challenge that required a lot of thought, and changes in previous parts of the codebase. Some of the AST nodes needed to store some extra information, and thus the parser had to be partially revisited.

6.1 Parser Changes

Although functional, array usage is quite restricted. For example, due to the definition of the grammar, expressions arent't allowed to be used as array elements. In other words, the following variable declaration would be syntactically incorrect:

```
let a:  int[4] = [1+2, 3, 4, 5];
```

The reason for this is that the grammar doesn't allow for expressions to be used as array elements. This is a limitation that could be easily fixed by adding a new rule to the grammar, but it was decided to leave it as is for the sake of simplicity.

Furthermore, it was noted that unlike `Ints`, `Bools`, `Colours` and `Floats`, the grammar doesn't offer a `Literal` variant for arrays. In a function declaration, this would again be syntactically incorrect:

```
fun rot_arr() -> int[4] {
    let a: int[4] = [1, 2, 3, 4];
    return [a[1], a[2], a[3], a[0]];
}
```

6.2 Semantic Analysis Changes

The semantic analysis was also updated to support arrays. The new type variant in the `Type` enum

```
Array(Box<Type>, usize)
```

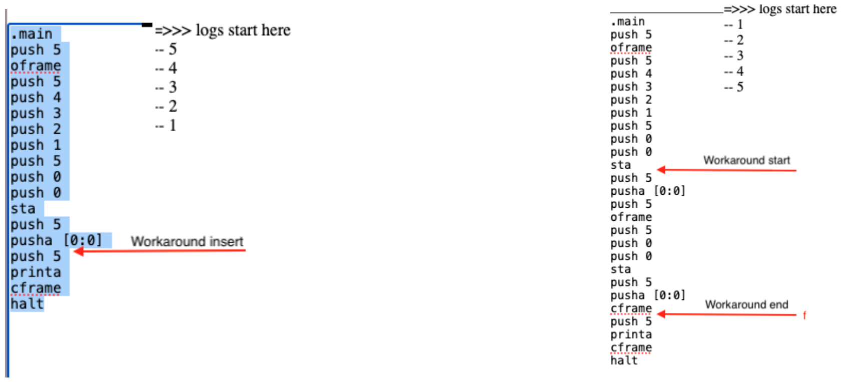
was introduced to represent arrays. Note that in a similar fashion to the `AstNode` enum, we have to wrap the type in a `Box` to avoid infinite size of the `Type` enum. Theoretically, this would allow the support for multi-dimensional arrays in our type system, but that would require massive changes starting again from the parser.

This changes allows us to hook up the type checker seamlessly, to ensure that when arrays are passed to functions as parameters, these are checked to be of the same type and size as the expected parameter.

6.3 Code Generation Changes

The code generation was also updated to support arrays. The main changes consisted of adding the new instructions, namely `StoreArray`, `PushArray(MemoryLocation)`, `PushOffsetFromOpS(MemoryLocation)`, `PrintArray` and `ReturnArray`. The latter however, although available as instruction in the VM, was not used. Even though clearly, we have seen that functions can return arrays (although they have to be declared temporarily inside and mutated only through direct array accesses), a slight workaround has been implemented to achieve **properly ordered** that happens to fix other issues that arise when referencing arrays. It was noted that some VM instructions are inconsistent with the ordering in the storage and loading of arrays from the operand stack.

Thus, when traversing the AST and finding an identifier which points to an array, the following workaround occurs.



```
.main
push 5
oframe
push 5
push 4
push 3
push 2
push 1
push 5
push 0
push 0
sta
push 5
pusha [0:0]
push 5
printa
cframe
halt
```

```
=>>> logs start here
-- 5
-- 4
-- 3
-- 2
-- 1
```

(a) Array printing without the workaround

```
.main
push 5
oframe
push 5
push 4
push 3
push 2
push 1
push 5
push 0
push 0
sta
push 5
pusha [0:0]
push 5
oframe
push 5
push 0
push 0
sta
push 5
pusha [0:0]
cframe
push 5
printa
cframe
halt
```

```
=>>> logs start here
-- 1
-- 2
-- 3
-- 4
-- 5
```

(b) Array printing with the workaround

Figure 7: Inconsistent pusha and printa behaviour

In essence, when we find an identifier that points to an array, and call PushArray, the contents of the array are actually pushed in reverse order. Calling PrintArray reveals this. The workaround consists of creating a new temporary frame with as much space as the array size, and then store the array in the frame. Subsequently the array is pushed back to the operand stack, but this time in the correct order. The frame is then closed and code generation continues as usual.

Like this, wherever we use arrays, we have the guarantee that the array is pushed in the correct order, and that it can be used anywhere as expected, even when accessing it.

7 Testing Framework

7.1 Unit Testing

During the early stages of development, such as the lexer and parser, most of the testing was done through unit tests, first with smaller cases, and then with larger cases.

7.2 Integration Testing

After the lexer and parser were completed, integration testing was done to ensure that the lexer and parser worked together correctly. This was done by running the CLI interfce, one stage at a time, on small, simple prorgrams.

Nearing the end of the project, I decided that it would be useful to create a more robust testing framework to test the entire compiler, as well as the compiled code on the VM. Using the Python library Playwright I created a script that opens the VM website in a browser (in headless mode) and runs the compiled code. The script then gets the output of the log, transforms it into an array of its lines, and removes the three '>' symbols at the beginning of each log entry.

In Python then, we are free to use this array to check against the expected values, through assertions, and we print out the results of the tests.

7.3 Results

Variable declarations

ParL Program:

```
let a: int = 5;
let b: colour = #000000;
let c: float  = 1.23;

let d: int[5] = [1, 2, 3, 4, 5];

__print a;
__print b;
__print c;
__print d;

{
    let a: int = 1;
    __print a;
}

__print a;
```

AST:

```
Program
  VarDec
    Identifier: a
    Type: int
    Expression: 5
  VarDec
    Identifier: b
    Type: colour
    Expression: #000000
  VarDec
    Identifier: c
    Type: float
    Expression: 1.23
  VarDecArray
    Identifier: d
    Element Type: int
    Size: 5
    Elements: 1, 2, 3, 4, 5
  Print
    Expression: a
  Print
    Expression: b
  Print
    Expression: c
  Print
    Expression: d
  Block
    VarDec
      Identifier: a
      Type: int
      Expression: 1
    Print
      Expression: a
  Print
    Expression: a
```

Result: Output successfuly matched with

```
['5', '0', '1.23', '1', '2', '3', '4', '5', '1', '5']
```

Variable Assignment

ParL Program:

```
let a: int = 5;
let b: colour = #000000;
let c: float  = 1.23;
let d: int[5] = [1, 2, 3, 4, 5];

__print a;
__print b;
__print c;
__print d[1];

a = 2;
b = #FFFFFF;
c = 3.45;
d[1] = 1;

__print a;
__print b;
__print c;
__print d[0];
```

AST:

```
Program
  VarDec
    Identifier: a
    Type: int
    Expression: 5
  VarDec
    Identifier: b
    Type: colour
    Expression: #000000
  VarDec
    Identifier: c
    Type: float
    Expression: 1.23
  VarDecArray
    Identifier: d
    Element Type: int
    Size: 5
    Elements: 1, 2, 3, 4, 5
  Print
    Expression: a
  Print
    Expression: b
  Print
    Expression: c
  Print
    Expression: ArrayAccess
        Identifier: d
        Index: 1
  Assignment
    Identifier: a
    Expression: 2
  Assignment
    Identifier: b
    Expression: #FFFFFF
  Assignment
    Identifier: c
    Expression: 3.45
  Assignment
    Identifier: d
    Expression: 1
    Index: 1
  Print
    Expression: a
  Print
    Expression: b
  Print
    Expression: c
```

```
Print
  Expression: ArrayAccess
      Identifier: d
      Index: 0
```

Result:

Output successfuly matched with

```
["5", "0", "1.23", "2", "2", "16777215", "3.45", "1"]
```

Control Flows

ParL Program:

```
let loop_max: int = 5;

for (let i: int = 0; i < loop_max; i = i + 1) {
    __print i;
}

if (1 < 2) {
    __print 1;
} else {
    __print 0;
}

if (1 > 2) {
    __print 1;
} else {
    __print 0;
}


let i: int = 0;
while (i < loop_max) {
    __print i;
    i = i + 1;
}
```

AST:

```
Program
  VarDec
    Identifier: a
    Type: int
    Expression: 5
  VarDec
    Identifier: b
    Type: colour
    Expression: #000000
  VarDec
    Identifier: c
    Type: float
    Expression: 1.23
  VarDecArray
    Identifier: d
    Element Type: int
    Size: 5
    Elements:
      1, 2, 3, 4, 5,
  Print
    Expression: a
  Print
    Expression: b
  Print
    Expression: c
  Print
    Expression: ArrayAccess
        Identifier: d
        Index: 1
  Assignment
    Identifier: a
    Expression: 2
  Assignment
    Identifier: b
    Expression: #FFFFFF
  Assignment
    Identifier: c
    Expression: 3.45
  Assignment
    Identifier: d
    Expression: 1
    Index: 1
  Print
    Expression: a
  Print
    Expression: b
  Print
    Expression: c
  Print
    Expression: ArrayAccess
        Identifier: d
        Index: 0
```

Result:

Output successfuly matched with

```
["0", "1", "2", "3", "4", "1", "0", "0", "1", "2", "3", "4"]
```

Functions

ParL Program:

```
fun add(a: int, b: int) -> int {
    return a + b;
}

let result: int = add(5, 10);

__print result;

fun some_func(a: int[2]) -> int[2] {
    let b: int[2] = [1,3];
    return b;
}

let a: int[2] = [1, 2];

__print some_func(a);


fun mixed_parameters(a: int, b: float, c: colour) -> bool {
    __print a;
    __print b;
    __print c;

    return true;
}

__print mixed_parameters(5, 1.23, #000000);

fun array_parameters(a: int[5]) -> int {

    let sum: int = 0;

    for (let i: int = 0; i < 5; i = i + 1) {
        __print a[i];
        sum = sum + a[i];
    }

    return sum;
}

let arr: int[5] = [5, 1, 2, 3, 4];

__print array_parameters(arr);
```

AST:

```
Program
  FunctionDecl
    Identifier: add
    Params: a: int, b: int,
    Return Type: int
    Block: Block
      Return
        Expression: (a + b)
  VarDec
    Identifier: result
    Type: int
    Expression: add(5, 10)
  Print
    Expression: result
  FunctionDecl
    Identifier: some_func
    Params: a: int[2],
    Return Type: int[2]
    Block: Block
      VarDecArray
        Identifier: b
        Element Type: int
        Size: 2
        Elements:
          1, 3,
      Return
        Expression: b
  VarDecArray
    Identifier: a
    Element Type: int
    Size: 2
    Elements:
      1, 2,
  Print
    Expression: some_func(a)
  FunctionDecl
    Identifier: mixed_parameters
    Params: a: int, b: float, c: colour,
    Return Type: bool
    Block: Block
      Print
        Expression: a
      Print
        Expression: b
      Print
        Expression: c
      Return
        Expression: true
  Print
    Expression: mixed_parameters(5, 1.23, #000000)
  FunctionDecl
    Identifier: array_parameters
    Params: a: int[5],
    Return Type: int
    Block: Block
      VarDec
        Identifier: sum
        Type: int
        Expression: 0
      For
        Initializer: VarDec
          Identifier: i
          Type: int
          Expression: 0
        Condition: (i < 5)
        Increment:
          Assignment
            Identifier: i
            Expression: (i + 1)

        Body: Block
          Print
            Expression: ArrayAccess
                Identifier: a
                Index: i
          Assignment
            Identifier: sum
            Expression: (sum + ArrayAccess
                Identifier: a
                Index: i)
      Return
        Expression: sum
  VarDecArray
    Identifier: arr
    Element Type: int
    Size: 5
    Elements:
      5, 1, 2, 3, 4,
  Print
    Expression: array_parameters(arr)
```

Result:

Output successfuly matched with

```
["15", "1", "3", "5", "1.23", "0", "1", "5", "1", "2", "3", "4", "15"]
```

Expressions

ParL Program:

```
let a: int = 5;
let b: int = 10;

__print (a + b) == 20;
__print ((((a + b) * 2) / 3) - 1) == 9;
__print (a + b) <= 20 and a - b <= 0;
__print ((12 + 5) * (8 - 3) + (18 / 2)) - ((25 - 5) / (4 + 1));
```

AST:

```
Program
  VarDec
    Identifier: a
    Type: int
    Expression: 5
  VarDec
    Identifier: b
    Type: int
    Expression: 10
  Print
    Expression: (((a + b)) == 20)
  Print
    Expression: (((((((((a + b)) * 2)) / 3)) - 1)) == 9)
  Print
    Expression: ((((a + b)) <= 20) and ((a - b) <= 0))
  Print
    Expression: ((((((12 + 5)) * ((8 - 3))) + ((18 / 2)))) - ((((25 - 5)) / ((4 + 1)))))
```

Result:

Output successfuly matched with

```
["0", "1", "1", "90"]
```

Pad Read/Write

ParL Program:

```
let c1: colour = #213B5F;
let c2: colour = #FFA400;

let x1: int = 5;
let y1: int = 10;

let x2: int = 20;
let y2: int = 30;

__write x1, y1, c1;

__print __read x1, y1;

__write_box x1, y1, x2, y2, c2;

__print __read x1 + 3, y1 + 3;
```

AST:

```
Program
    VarDec
      Identifier: c1
      Type: colour
      Expression: #213B5F
    VarDec
      Identifier: c2
      Type: colour
      Expression: #FFA400
    VarDec
      Identifier: x1
      Type: int
      Expression: 5
    VarDec
      Identifier: y1
      Type: int
      Expression: 10
    VarDec
      Identifier: x2
      Type: int
      Expression: 20
    VarDec
      Identifier: y2
      Type: int
      Expression: 30
    __write x1, y1, c1;
    Print
```

```
  Expression: __read x1, y1
__write_box x1, y1, x2, y2, c2;
Print
  Expression: __read (x1 + 3), (y1 + 3)
```

Result:

Output successfuly matched with

```
["2177887", "16753664"]
```

References

[1] R. Nystrom. *Crafting Interpreters*. Genever Benning, 2021. ISBN: 9780990582939. URL: https://books.google.com.mt/books?id=ySOBzgEACAAJ.

[2] *Scope and Shadowing - Rust By Example — doc.rust-lang.org*. https://doc.rust-lang.org/rust-by-example/variable_bindings/scope.html. [Accessed 26-05-2024].

[3] Linda Torczon and Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 86–71. ISBN: 012088478X.

A Lexer DFSA

```
pub fn new(input: &str, file: &Path, dfsa: Option<Dfsa>) -> Self {
    let mut dfsa_builder = DfsaBuilder::new();

    match dfsa {
        Some(dfsa) => Lexer {
            buffer: B::new(input, file),
            dfsa,
        },
        None => {
            let dfsa = dfsa_builder
                .add_category(’a’..=’f’, Category::HexAndLetter)
                .add_category(’A’..=’F’, Category::HexAndLetter)
                .add_category(’g’..=’z’, Category::Letter)
                .add_category(’G’..=’Z’, Category::Letter)
                .add_category(’0’..=’9’, Category::Digit)
                .add_final_character_symbols(vec![
                    (’\n’, Category::Newline, TokenKind::Newline),
                    (’{’, Category::LBrace, TokenKind::LBrace),
                    (’}’, Category::RBrace, TokenKind::RBrace),
                    (’(’, Category::LParen, TokenKind::LParen),
                    (’)’, Category::RParen, TokenKind::RParen),
                    (’[’, Category::LBracket, TokenKind::LBracket),
                    (’]’, Category::RBracket, TokenKind::RBracket),
                    (’;’, Category::Semicolon, TokenKind::Semicolon),
                    (’:’, Category::Colon, TokenKind::Colon),
                    (’+’, Category::Plus, TokenKind::Plus),
                    (’*’, Category::Asterisk, TokenKind::Multiply),
                    (’,’, Category::Comma, TokenKind::Comma),
                    (’\0’, Category::Eof, TokenKind::EndOfFile),
                    (’%’, Category::Percent, TokenKind::Mod),
                ])
                .add_whitespace_logic()
                .add_comment_functionality()
                .add_multi_char_rel_ops()
                .add_identifier_logic()
                .add_number_logic()
                .build();

            Lexer {
                buffer: B::new(input, file),
                dfsa,
            }
        }
    }
}
```

B Parser AST Node Enum

```
pub type AstNodePtr = Box<AstNode>;

#[derive(Debug)]
pub enum AstNode {
    Program {
        statements: Vec<AstNode>,
    },
    VarDec {
        identifier: Token,
        r#type: Token,
        expression: AstNodePtr,
    },
    Block {
        statements: Vec<AstNode>,
    },
    Expression {
        casted_type: Option<Token>,
        expr: AstNodePtr,
    },
    SubExpression {
        bin_op: AstNodePtr,
    },
    UnaryOp {
        operator: Token,
        expr: AstNodePtr,
    },
    BinOp {
        left: AstNodePtr,
        operator: Token,
        right: AstNodePtr,
    },
    PadWidth,
    PadRandI {
        upper_bound: AstNodePtr,
    },
    PadHeight,
    PadRead {
        x: AstNodePtr,
        y: AstNodePtr,
    },
```

```rust
    IntLiteral(Token),
    FloatLiteral(Token),
    BoolLiteral(Token),
    ColourLiteral(Token),
    FunctionCall {
        identifier: Token,
        args: Vec<AstNode>,
    },
    ActualParams {
        params: Vec<AstNode>,
    },
    Delay {
        expression: AstNodePtr,
    },
    Return {
        expression: AstNodePtr,
    },
    PadWriteBox {
        loc_x: AstNodePtr,
        loc_y: AstNodePtr,
        width: AstNodePtr,
        height: AstNodePtr,
        colour: AstNodePtr,
    },
    PadWrite {
        loc_x: AstNodePtr,
        loc_y: AstNodePtr,
        colour: AstNodePtr,
    },
    Identifier {
        token: Token,
    },
    If {
        condition: AstNodePtr,
        if_true: AstNodePtr,
        if_false: Option<AstNodePtr>,
    },
    For {
        initializer: Option<AstNodePtr>,
        condition: AstNodePtr,
        increment: Option<AstNodePtr>,
        body: AstNodePtr,
    },
    While {
        condition: AstNodePtr,
        body: AstNodePtr,
    },
    FormalParam {
        identifier: Token,
        param_type: Token,
        index: Option<Token>,
    },
    FunctionDecl {
        identifier: Token,
        params: Vec<AstNode>,
        return_type: Type,
        block: AstNodePtr,
    },
    Print {
        expression: AstNodePtr,
    },
    Assignment {
        identifier: Token,
        expression: AstNodePtr,
        index: Option<AstNodePtr>,
    },
    PadClear {
        expr: AstNodePtr,
    },
    VarDecArray {
        identifier: Token,
        element_type: Token,
        size: usize,
        elements: Vec<AstNode>,
    },
    ArrayAccess {
        identifier: Token,
        index: AstNodePtr,
    },
    EndOfFile,
}
```