Compiling ParL to PArIR in Rust

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

Giorgio Grigolo - 0418803L

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. It was implemented incrementally, as to ensure each component can be run in isolation, and is working correctly before moving on to the next one.

1 Project Structure

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* [2]. 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 the compiler.

2.1 DFSA Builder

Of course, 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) grew 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:

```
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

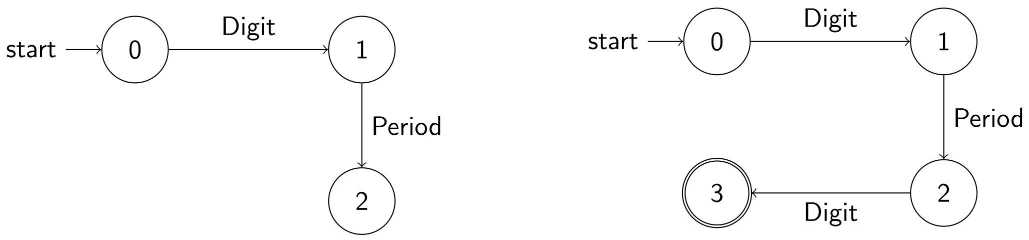
(a) The DFSA before adding calling `add_transition`   (b) The DFSA after adding calling `add_transition`

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.

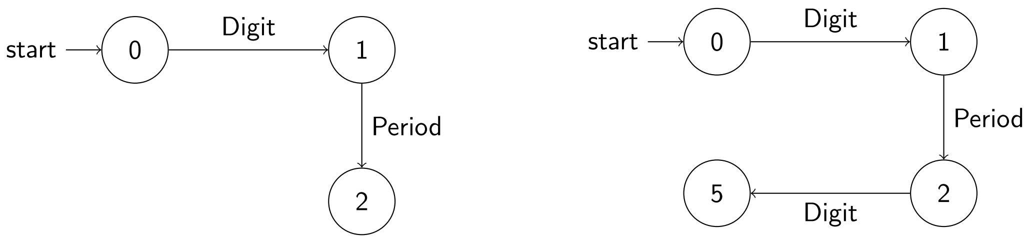
If we run `auto_add_transition(2, Category::Digit, None, Category::Float)`, the resultant DFSA will be as follows:



(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.

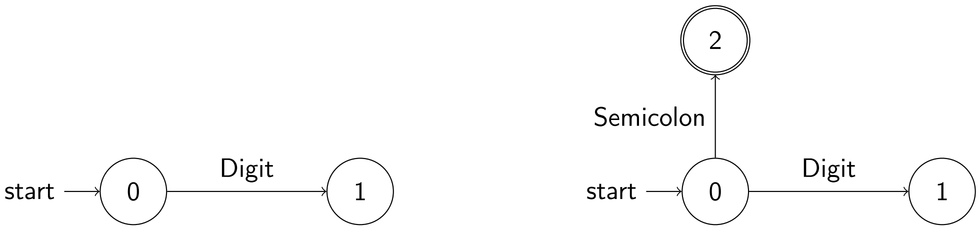


Next up, we have the function

```
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.

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.

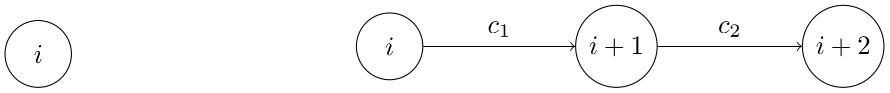
The `Transition` struct holds a reference to the `DfsaBuilder` that it was created from, the current state, the list of transitions that have been added, and the a list of final states with the token kind that they return.

```
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
```

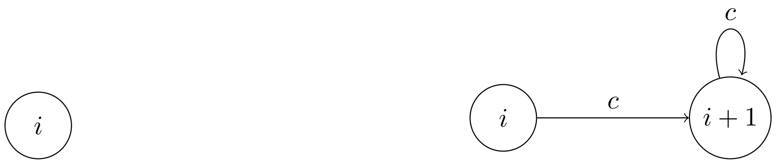
This method is used to just append a single category to the transition. Calling it multiple times, increments an internal pointer, that always points to the last state that was added. Thus, calling it multiple times creates a *chain* of states.



(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 $\star$ 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)`

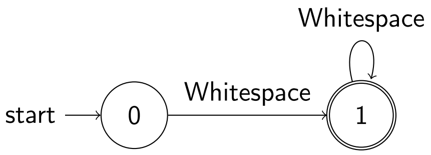
method, that simply marks the last state as an accepted state, and sets the token kind it returns to the one that was passed as an argument.

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:

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

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.

4 Semantic Analysis

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

```rust
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,
    },
    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,
}
```

References

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

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