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 nwas 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 poosition 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 functions that a struct implementing the `Stream` trait, which can be seen in Listing 2, are very similar to the ones proposed in the textbook *Engineering a Compiler* [1], as the lexer algorithm also follows the pseudo-code provided in the book.

```rust
pub trait Stream {
    fn new(input: &str, path: &Path) -> Self;
    fn rollback(&mut self);
    fn next_char(&mut self) -> char;
    fn get_line(&self) -> usize;
    fn get_col(&self) -> usize;
    fn get_input_pointer(&self) -> usize;
    fn is_eof(&self) -> bool;
    fn current_char(&self) -> char;
    fn file_path(&self) -> &str;
}
```

Listing 2: The `Stream` trait.

Of course, the `Lexer` that as 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.

2.1 DFSA Builder

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.

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

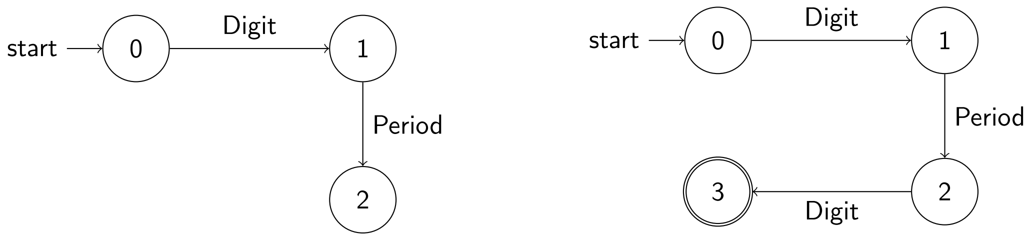
The next function is

```
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.

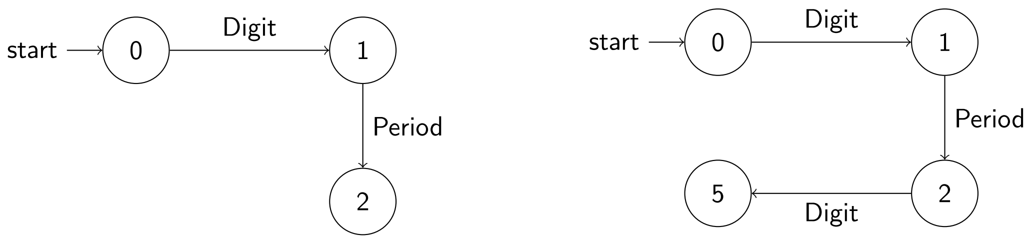
This is useful for adding transitions to the same state, as well as constructing slightly more complicated sequences of transitions, say for tokenizing a float. The `token_kind` parameter is used to set the token kind of the state, if it is an accepted state. Similarly to the `to` parameter, it can be `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.



References

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