# SIMPLE RUST COMPILER

Will Vipperman, Yarrick Dillard and Bohan Chen

```rust
impl Rule {
    new *
    pub fn new(lhs: char, rhs: &str) -> Self {
        Rule {
            lhs,
            rhs: rhs.to_string(),
        }
    }
```

```rust
impl Grammar {
    new *
    pub fn from_rules(rules: &[Rule]) -> Self {
        let mut non_terminal_string : String = String::new();
        let mut terminals_string : String = String::new();
        let start : char = rules[0].lhs;

        for rule : &Rule in rules {
            if !non_terminal_string.contains(rule.lhs) {
                non_terminal_string.push(rule.lhs);
            }
            for c : char in rule.rhs.chars() {
                if c.is_ascii_uppercase() {
                    if !non_terminal_string.contains(c) {
                        non_terminal_string.push(c);
                    }
                }
                else if !terminals_string.contains(c) {
                    terminals_string.push(c);
                }
            }
        }
```

```rust
impl Sentential {
    new *
    pub fn new_initial(grammar: &Grammar) -> Sentential {
        Sentential {
            form: grammar.start.to_string(),
            nt_idx: 0,
        }
    }
```

```rust
impl Derivation {
    new *
    pub fn new(grammar: &Grammar) -> Derivation {
        let mut new_sent : Sentential = Sentential::new_initial(grammar);
        Derivation {
            steps: vec![(-1, new_sent)],
        }
    }
```

```rust
pub enum Token {
    PARENS_L, PARENS_R,
    BRACKET_L, BRACKET_R,
    BRACE_L, BRACE_R,
    POINT, COMMA, COLON, SEMICOLON, ARROW_R,
    // arithmetic operators
    ADD, SUB, MUL, DIV,
    ADD_ASSIGN, SUB_ASSIGN, MUL_ASSIGN, DIV_ASSIGN,
    // relational operators
    EQ,     // equal
    LT,     // less than
    GT,     // greater than
    NEQ,    // not equal
    NLT,    // not less than == greater than or equal
    NGT,    // not greater than == less than or equal
    // logical operators
    NOT, AND, OR,
    // other operators
    ASSIGN, AMPERSAND,
    // keywords
    MUT, FUNC, LET, IF, ELSE, ELSE_IF, WHILE, PRINT, RETURN,

    // types
    TYPE_INT32, TYPE_FLT32, TYPE_CHAR,

    // literals
    ID(String),
    LIT_INT32(i32),
    LIT_FLT32(f32),
    LIT_CHAR(char),
    LIT_STRING(String),
    TRUE,
    FALSE,
```

```rust
    // special characters
    LINEBREAK, EOI, ERROR,

    // Meta tokens for AST
    META_PROGRAM, META_FUNC, META_PARAM_LIST, META_PARAM,
    META_LET, META_RETURN, META_IF, META_ELSE_IF,
    META_BLOCK, META_VOID, META_INFER,
    META_ASSIGN, META_CALL, META_PRINT, META_WHILE,

    // Keywords.
    READ, WRITE
```

```rust
pub enum LexerState {
    Start,
    Operation,
    Stage,
    NumberInt,
    NumberFloat,
    NumberChar,
    NumberString,
    NumberArray,
    CharArray,
    StringArray,
    CharLit,
    StringLit,
    Return
}
```

```rust
pub fn advance(&mut self) -> Option<Token> {
    self.token = None;

    while self.input_pos < self.input_string.len() {
        let current_char: char = self.input_string.as_bytes()[self.input_pos] as char;
        // let current_char: char = self.next_char().unwrap();
        match self.state {
            LexerState::Start => {
                if vec!['\n'].contains(&current_char) {
                    self.input_pos += 1;
                    let token = Token::LINEBREAK;
                    self.token = Some(token.clone());
                    return Some(token);
                }
                if current_char.is_whitespace() {
                    self.input_pos += 1;
                    continue;
                }

                if self.input_pos >= self.input_string.len() {
                    let token = Token::EOI;
                    self.token = Some(token.clone());
                    return Some(token);
                }
                self.buffer_string.clear();
                if current_char.is_alphabetic() || current_char == '_' || current_char.is_digit( radix: 10) {
                    self.input_pos += 1;
                    self.buffer_string.push(current_char);
                    self.state = if current_char.is_digit( radix: 10) { LexerState::NumberInt } else { LexerState::Operation };
                    continue;
                }
```

```rust
if vec!['(', ')', '[', ']', '{', '}', ',', ':', ';', '&', '|'].contains(&current_char) {
    let token: Token;
    match current_char {
        '(' => token = Token::PARENS_L,
        ')' => token = Token::PARENS_R,
        '[' => token = Token::BRACKET_L,
        ']' => token = Token::BRACKET_R,
        '{' => token = Token::BRACE_L,
        '}' => token = Token::BRACE_R,
        ',' => token = Token::COMMA,
        ':' => token = Token::COLON,
        ';' => token = Token::SEMICOLON,
        '&' => {
            if self.peek_char() == Some('&') {
                self.input_pos += 1;
                token = Token::AND;
            } else {
                token = Token::AMPERSAND;
            }
        },
        '|' => {
            if self.peek_char() == Some('|') {
                self.input_pos += 1;
                token = Token::OR;
            } else {
                token = Token::ERROR;
            }
        }
        _ => token = Token::ID(current_char.to_string()),
    }
```

```rust
use std::mem::discriminant;
use crate::hw_assignment_3::*;
use std::rc::Rc;
use crate::token::TCode;

2 usages
const INDENT : usize = 2;

8 usages
pub struct Parser {
    lexer: Lexer,
    pub(crate) indent: usize,
}
```

```rust
impl Parser {

    pub fn new(mut lexer: Lexer) -> Parser {
        lexer.advance();
        Parser {
            lexer,
            indent: 0,
        }
    }
    pub fn analyze(&mut self) -> MTree {
        self.indent = 0;
        let tree : MTree = self.parse_program();
        self.expect(Token::EOI);
        tree
    }
}
```

```rust
use std::cell::RefCell;
use hw_assignment_2::*;
use hw_assignment_3::*;
use hw_assignment_4::*;
```

```rust
fn print_help_for(command: &str) {
    match command {
        "help" => {
            println!("help [command]: \n- prints help info for commands\n");
        }
        "print" => {
            println!("print <file> [--numbered]: \n- prints out the contents of a
        }
        "list" => {
            println!("list commands OR list rules OR list tokens: \n- prints the l
        }
        "derive" => {
            println!("derive random [limit of derivation steps]: \n- generates and
            println!("derive <int-list> [sequence of numbers]: \n- manually genera
        }
        "example" => {
            println!("example: \n- runs an example demonstrating the grammar\n");
        }
        "tokenize" => {
            println!("tokenize <file>: \n- tokenizes the input from a file and the
        }
        "parse" => {
            println!("parse <file>: \n- tokenizes & parses the input from a file a
        }
        _ => {
            println!("Unknown command: {}.\n", command);
        }
    }
}
```

```rust
fn ConvertTokenToTCode(bc_token: hw_assignment_3::Token) -> TCode {
    match bc_token {
        hw_assignment_3::Token::PARENS_L => {
            TCode::PAREN_L
        }
        hw_assignment_3::Token::PARENS_R => {
            TCode::PAREN_R
        }
        hw_assignment_3::Token::BRACKET_L => {
            TCode::BRACKET_L
        }
        hw_assignment_3::Token::BRACKET_R => {
            TCode::BRACKET_R
        }
        hw_assignment_3::Token::BRACE_L => {
            TCode::BRACE_L
        }
        hw_assignment_3::Token::BRACE_R => {
            TCode::BRACE_R
        }
        hw_assignment_3::Token::POINT => {
            TCode::POINT
        }
        hw_assignment_3::Token::COMMA => {
            TCode::COMMA
        }
        hw_assignment_3::Token::COLON => {
            TCode::COLON
        }
```

```rust
fn ConvertbcChildrenToOhlChildren(bc_children: Vec<Rc<hw_assignment_4::MTree>>) -> Vec<Rc<MTree>> {
    let ohl_children: Vec<Rc<MTree>> = vec![];
    for bc_child : Rc<MTree> in bc_children {
        let ohl_child = MTree {
            token: Token::from(ConvertTokenToTCode(bc_child.token.clone())),
            children: vec![],
        };
    }

    ohl_children
}
```

# Questions?