PRÁCTICA 4

Abel Eduardo Robles Lázaro

17 DE ABRIL DE 2023

UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Desarrollo:

Se pretende establecer las bases de un analizador sintáctico al analizar la gramática para sumas, restas y multiplicaciones de operaciones con un solo dígito. Junto con reglas semánticas para calcular la notación posfija. Para realizar este analizador, se requiere de un autómata de pila, es decir un programa capaz de desplazarse por la cadena de entrada, modificarla y que para ello requiere una pila de memoria.

Gramática

G = { V, Σ, $Q0$, P }
V = { 0,1,2,3,4,5,6,7,8,9, −, +,/,∗, (,) }
∑ = { expr, term, factor, digit, rest_term, rest_expr }
$Q0$ = $expr$
P = {
expr → term rest_expr
rest_expr → + term rest_expr
rest_expr → - term rest_expr
rest_expr → ε
term → factor rest_term
rest_term → / factor rest_term
rest_term → * factor rest_term
rest_term → ε
factor → (expr)
factor → id
factor → number
id → _ rest_id
id → letra rest_id
rest_id → letra rest_id
rest_id → digit rest_id
rest_id → _ rest_id
rest_id → ε
letra → A...z
number → digit rest_num
rest_num → digit rest_num
rest_num → ε
digit → 0...9
}

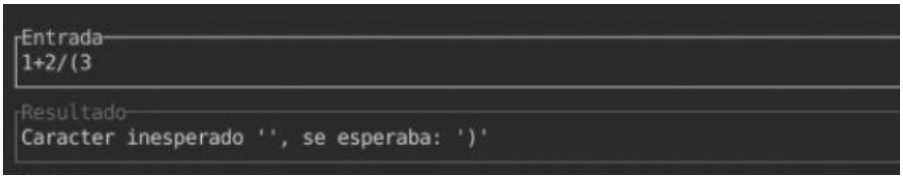
Capturas

Una entrada simple de 1+2:

También se incluye en los resultados la cadena en postfijo y en prefijo.

Un segundo ejemplo con una cadena más complicada:

Un ejemplo con una cadena inválida:



Código fuente:

Main.rs:

```
use std::{env::args, fs};

use semantic::SemanticAnalyzer;

pub mod lexic;
pub mod production;
pub mod semantic;
pub mod sintactic;
pub mod symbols;
pub mod token;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let path = args().nth(1).expect("No file path given");
    let contets = fs::read_to_string(path)?;
    let mut semantic = SemanticAnalyzer::new();
    let res = semantic.parse(&contets)?;
    println!("{}", res);
    Ok(())
}
```

Analyzer.rs

```
use super::tree::TreeItem;
use std::{char, error, fmt};
#[derive(Debug, Clone, Default)]
pub struct AnalyzerError {
    character: char,
    expected: String,
}
impl AnalyzerError {
    pub fn new(character: char, expected: String) -> Self {
        AnalyzerError {
            character,
            expected,
        }
    }
}
impl error::Error for AnalyzerError {}
impl fmt::Display for AnalyzerError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(
            f,
            "Caracter inesperado '{}', se esperaba: '{}'",
            self.character, self.expected
        )
    }
}
#[derive(Debug, Clone, Default)]
pub struct Analyzed {
    pub result: f32,
    pub postfix: String,
    pub prefix: String,
```

```
    pub tree: TreeItem,
}
pub type AnalyzerResult = Result<Analyzed, AnalyzerError>;
#[derive(Debug, Clone)]
pub struct Analyzer {
    input: String,
    current: char,
}
impl Analyzer {
    pub fn new(input: &String) -> Self {
        let mut input_copy = input.clone();
        let current = input_copy.remove(0);
        Analyzer {
            input: input_copy,
            current,
        }
    }
    pub fn check_and_next(&mut self, character: char) -> Result<(),
AnalyzerError> {
        if self.current != character {
            return Err(AnalyzerError::new(self.current,
String::from(character)));
        }
        if self.input.len() > 0 {
            self.current = self.input.remove(0);
        } else {
            self.current = '\0';
        }
        Ok(())
    }
    pub fn analyze(&mut self) -> AnalyzerResult {
        let res = self.expr();
        if self.current != '\0' {
            return Err(AnalyzerError::new(self.current,
String::from('\0')));
        }
        res
    }
    pub fn expr(&mut self) -> AnalyzerResult {
        let term = self.term()?;
        let mut res = self.rest_expr(&term)?;
        res.tree = TreeItem {
            root: "expr".to_string(),
            items: vec![term.tree, res.tree],
        };
        Ok(res)
    }
    pub fn rest_expr(&mut self, analyzed: &Analyzed) -> AnalyzerResult
{
        let root = "rest_expr".to_string();
        match self.current {
            '+' => {
                self.check_and_next('+')?;
                let term = self.term()?;
```

```
                let mut partial = Analyzed {
                    result: analyzed.result + term.result,
                    postfix: format!("{} {} +", analyzed.postfix,
term.postfix),
                    prefix: format!("+ {} {}", analyzed.prefix,
term.prefix),
                    tree: TreeItem {
                        root,
                        items: vec![TreeItem::new("+".to_string()),
term.tree],
                    },
                };
                let mut res = self.rest_expr(&partial)?;
                partial.tree.items.push(res.tree);
                res.tree = partial.tree;
                Ok(res)
            }
            '-' => {
                self.check_and_next('-')?;
                let term = self.term()?;
                let mut partial = Analyzed {
                    result: analyzed.result - term.result,
                    postfix: format!("{} {} -", analyzed.postfix,
term.postfix),
                    prefix: format!("- {} {} ", analyzed.prefix,
term.prefix),
                    tree: TreeItem {
                        root,
                        items: vec![TreeItem::new("-".to_string()),
term.tree],
                    },
                };
                let mut res = self.rest_expr(&partial)?;
                partial.tree.items.push(res.tree);
                res.tree = partial.tree;
                Ok(res)
            }
            _ => {
                let mut res = analyzed.clone();
                res.tree = TreeItem {
                    root,
                    items: vec![TreeItem::new(String::from("ε"))],
                };
                Ok(res)
            }
        }
    }
    pub fn term(&mut self) -> AnalyzerResult {
        let factor = self.factor()?;
        let mut res = self.rest_term(&factor)?;
        res.tree = TreeItem {
            root: "term".to_string(),
            items: vec![factor.tree, res.tree],
        };
```

```
        Ok(res)
    }
    pub fn rest_term(&mut self, analyzed: &Analyzed) -> AnalyzerResult
{
        let root = "rest_term".to_string();
        match self.current {
            '*' => {
                self.check_and_next('*')?;
                let factor = self.factor()?;
                let mut partial = Analyzed {
                    result: analyzed.result * factor.result,
                    postfix: format!("{} {} *", analyzed.postfix,
factor.postfix),
                    prefix: format!("* {} {} ", analyzed.prefix,
factor.prefix),
                    tree: TreeItem {
                        root,
                        items: vec![TreeItem::new("*".to_string()),
factor.tree],
                    },
                };
                let mut res = self.rest_term(&partial)?;
                partial.tree.items.push(res.tree);
                res.tree = partial.tree;
                Ok(res)
            }
            '/' => {
                self.check_and_next('/')?;
                let factor = self.factor()?;
                let mut partial = Analyzed {
                    result: analyzed.result / factor.result,
                    postfix: format!("{} {} /", analyzed.postfix,
factor.postfix),
                    prefix: format!("/ {} {}", analyzed.prefix,
factor.prefix),
                    tree: TreeItem {
                        root,
                        items: vec![TreeItem::new("/".to_string()),
factor.tree],
                    },
                };
                let mut res = self.rest_term(&partial)?;
                partial.tree.items.push(res.tree);
                res.tree = partial.tree;
                Ok(res)
            }
            _ => {
                let mut res = analyzed.clone();
                res.tree = TreeItem {
                    root,
                    items: vec![TreeItem::new(String::from("ε"))],
                };
                Ok(res)
            }
```

```
            }
        }
        pub fn factor(&mut self) -> AnalyzerResult {
            if self.current == '(' {
                self.check_and_next('(')?;
                let mut analyzed = self.expr()?;
                self.check_and_next(')')?;
                analyzed.tree = TreeItem {
                    root: String::from("factor"),
                    items: vec![
                        TreeItem::new("(".to_string()),
                        analyzed.tree,
                        TreeItem::new(")".to_string()),
                    ],
                };
                return Ok(analyzed);
            }
            let mut factor = self.digit()?;
            factor.tree = TreeItem {
                root: String::from("factor"),
                items: vec![factor.tree],
            };
            Ok(factor)
        }
        pub fn digit(&mut self) -> AnalyzerResult {
            let parsed = match self.current.to_digit(10) {
                Some(digit) => digit,
                None => {
                    return Err(AnalyzerError::new(
                        self.current,
                        String::from("digito (0..9)"),
                    ))
                }
            };
            let analyzed = Analyzed {
                result: parsed as f32,
                postfix: self.current.to_string(),
                prefix: self.current.to_string(),
                tree: TreeItem {
                    root: "digit".to_string(),
                    items: vec![TreeItem::new(self.current.to_string())],
                },
            };
            self.check_and_next(self.current)?;
            Ok(analyzed)
        }
    }
```

Conclusiones:

Una de las conclusiones más importantes es que este tipo de analizador no funciona con una gramática que tenga recursión por la izquierda ya que requeriría empezar la cadena por el final o condiciones más complejas, que si bien no son cambios complejos a gran escala podrían resultar molestos.

Otra cosa importante a notar es que la arquitectura con la que está hecha el analizador, es decir, utilizando las funciones como estados, escala de forma muy pobre, como se puede observar en el código fuente, aun con unas pocas reglas en la gramática el código fue extenso y en cierta forma repetitivo, además de que en caso de tener una gran anidación en el árbol.