Лабораторная работа № 1.3 «Объектно-ориентированный лексический анализатор»

6 марта 2024 г.

Кирилл Киселёв, ИУ9-61Б

Цель работы

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

Индивидуальный вариант

- Идентификаторы: последовательности латинских букв и десятичных цифр, оканчивающиеся на цифру.
- Числовые литералы: непустые последовательности десятичных цифр, органиченные знаками «<» и «>».
- Операции: «<=», «=», «==».

Реализация

Лабораторная работа выполнялась на языке Rust

Файл `position.rs`

```rust
use std::{fmt::Display, iter::Peekable, str::Chars};

#[derive(Clone)]
pub struct Position {
    line: usize,
    pos: usize,
    index: usize,
    iter: Peekable<Chars<'static>>,
}
```

```rust
impl Position {
    pub fn new(iter: Chars<'static>) -> Self {
        Self {
            line: 1,
            pos: 1,
            index: 0,
            iter: iter.peekable(),
        }
    }

    pub fn line(&self) -> usize {
        self.line
    }

    pub fn pos(&self) -> usize {
        self.pos
    }

    pub fn index(&self) -> usize {
        self.index
    }

    pub fn next(&mut self) -> Option<char> {
        if self.is_newline() {
            self.line += 1;
            self.pos = 1;
        } else {
            self.pos += 1;
        }

        self.index += 1;

        if let Some(c) = self.iter.next() {
            return Some(c);
        }

        None
    }

    pub fn cp(&mut self) -> Option<char> {
        if let Some(c) = self.iter.peek() {
            return Some(*c);
        }

        None
    }
```

```rust
    pub fn is_newline(&mut self) -> bool {
        if let Some(c) = self.iter.peek() {
            return c.eq(&'\n');
        }

        false
    }

    pub fn is_whitespace(&mut self) -> bool {
        if let Some(c) = self.iter.peek() {
            return c.is_whitespace();
        }

        false
    }

    pub fn is_alphabetic(&mut self) -> bool {
        if let Some(c) = self.iter.peek() {
            return c.is_alphabetic();
        }

        false
    }

    pub fn is_numeric(&mut self) -> bool {
        if let Some(c) = self.iter.peek() {
            return c.is_numeric();
        }

        false
    }

    pub fn is_alphanumeric(&mut self) -> bool {
        if let Some(c) = self.iter.peek() {
            return c.is_alphanumeric();
        }

        false
    }
}

impl Display for Position {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "({}, {})", self.line, self.pos)
    }
```

```
}
```

Файл `token.rs`

```rust
use std::fmt::Display;

use crate::{domain::DomainTag, fragment::Fragment, position::Position};

pub trait Token: Display {
    fn get_domain_tag(&self) -> DomainTag;
    fn get_coords(&self) -> Fragment;
}

pub struct IdentToken {
    code: usize,
    coords: Fragment,
}

impl IdentToken {
    pub fn new(code: usize, starting: Position, following: Position) -> Self {
        Self {
            code,
            coords: Fragment::new(starting, following),
        }
    }
}

impl Token for IdentToken {
    fn get_domain_tag(&self) -> DomainTag {
        DomainTag::IDENT
    }

    fn get_coords(&self) -> Fragment {
        self.coords.clone()
    }
}

impl Display for IdentToken {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{} {}-{}: {}",
            DomainTag::IDENT,
            self.get_coords().starting(),
            self.get_coords().following(),
            self.code
        )
```

```rust
    }
}

pub struct OperationToken {
    value: String,
    coords: Fragment,
}

impl OperationToken {
    pub fn new(value: &str, starting: &Position, following: &Position) -> Self {
        Self {
            value: value.to_string(),
            coords: Fragment::new(starting.clone(), following.clone()),
        }
    }
}

impl Token for OperationToken {
    fn get_domain_tag(&self) -> DomainTag {
        DomainTag::OPERATION
    }

    fn get_coords(&self) -> Fragment {
        self.coords.clone()
    }
}

impl Display for OperationToken {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{} {}-{}: {}",
            DomainTag::OPERATION,
            self.get_coords().starting(),
            self.get_coords().following(),
            self.value
        )
    }
}

pub struct NumberToken {
    value: i128,
    coords: Fragment,
}

impl NumberToken {
```

```
    pub fn new(value: i128, starting: Position, following: Position) -> Self {
        Self {
            value,
            coords: Fragment::new(starting, following),
        }
    }
}

impl Token for NumberToken {
    fn get_domain_tag(&self) -> DomainTag {
        DomainTag::NUMBER
    }

    fn get_coords(&self) -> Fragment {
        self.coords.clone()
    }
}

impl Display for NumberToken {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{} {}-{}: {}",
            DomainTag::NUMBER,
            self.get_coords().starting(),
            self.get_coords().following(),
            self.value
        )
    }
}

pub struct EOPToken {
    coords: Fragment,
}

impl EOPToken {
    pub fn new(end: &Position) -> Self {
        Self {
            coords: Fragment::new(end.clone(), end.clone()),
        }
    }
}

impl Token for EOPToken {
    fn get_domain_tag(&self) -> DomainTag {
        DomainTag::EOP
```

```
    }

    fn get_coords(&self) -> Fragment {
        self.coords.clone()
    }
}

impl Display for EOPToken {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{} {}-{}",
            DomainTag::EOP,
            self.coords.starting(),
            self.coords.following()
        )
    }
}
```

Файл `compiler.rs`
```rust
use std::fmt::Display;

#[derive(Debug)]
pub struct Message {
    is_err: bool,
    message: String,
}

impl Message {
    pub fn new(is_err: bool, message: &str) -> Self {
        Self {
            is_err,
            message: message.to_string(),
        }
    }

    pub fn is_err(&self) -> bool {
        self.is_err
    }

    pub fn message(&self) -> &str {
        self.message.as_ref()
    }
}
```

```rust
impl Display for Message {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if self.is_err {
            return write!(f, "ERR: {}", self.message);
        }

        write!(f, "MSG: {}", self.message)
    }
}
```

Файл compiler.rs

```rust
use crate::position;

#[derive(Clone)]
pub struct Fragment {
    starting: position::Position,
    following: position::Position,
}

impl Fragment {
    pub fn new(starting: position::Position, following: position::Position) -> Self {
        Self {
            starting,
            following,
        }
    }

    pub fn starting(&self) -> position::Position {
        self.starting.clone()
    }

    pub fn following(&self) -> position::Position {
        self.following.clone()
    }
}
```

Файл compiler.rs

```rust
use std::collections::HashMap;

use crate::message::Message;
use crate::position::Position;

pub struct Compiler {
    names: Vec<String>,
    name_codes: HashMap<String, usize>,
    messages: Vec<(Position, Message)>,
```

```rust
}

impl Compiler {
    pub fn new() -> Self {
        Self {
            names: vec![],
            name_codes: HashMap::new(),
            messages: vec![],
        }
    }

    pub fn add_name(&mut self, name: &str) -> usize {
        if let Some(code) = self.name_codes.get(name) {
            return *code;
        }

        let code = self.name_codes.len();
        self.names.push(name.to_string());
        self.name_codes.insert(name.to_string(), code);

        code
    }

    pub fn get_name(&self, code: usize) -> &str {
        &self.names[code]
    }

    pub fn add_message(&mut self, is_err: bool, msg: &str, pos: Position) {
        self.messages.push((pos, Message::new(is_err, msg)));
    }

    pub fn messages(&self) -> &[(Position, Message)] {
        self.messages.as_ref()
    }

    pub fn names(&self) -> &[String] {
        self.names.as_ref()
    }
}

impl Default for Compiler {
    fn default() -> Self {
        Self::new()
    }
}
```

Файл `domain.rs`

```rust
use std::fmt::Display;

#[derive(Debug)]
pub enum DomainTag {
    IDENT,
    NUMBER,
    OPERATION,
    EOP,
}

impl Display for DomainTag {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            DomainTag::IDENT => write!(f, "IDENT"),
            DomainTag::NUMBER => write!(f, "NUMBER"),
            DomainTag::OPERATION => write!(f, "OPERATION"),
            DomainTag::EOP => write!(f, "EOP"),
        }
    }
}
```

Файл `scanner.rs`

```rust
use crate::position;

#[derive(Clone)]
pub struct Fragment {
    starting: position::Position,
    following: position::Position,
}

impl Fragment {
    pub fn new(starting: position::Position, following: position::Position) -> Self {
        Self {
            starting,
            following,
        }
    }

    pub fn starting(&self) -> position::Position {
        self.starting.clone()
    }

    pub fn following(&self) -> position::Position {
        self.following.clone()
```

```
    }
}
```

Файл `scanner.rs`

```rust
use crate::{
    compiler::Compiler,
    message::Message,
    position::Position,
    token::{self, IdentToken, OperationToken, Token},
};

pub struct Scanner {
    compiler: Compiler,
    cur: Position,
    recovering: bool,
}

impl Scanner {
    pub fn new(program: &'static str, compiler: Compiler) -> Self {
        Self {
            compiler,
            cur: Position::new(program.chars()),
            recovering: false,
        }
    }

    pub fn get_messages(&self) -> &[(Position, Message)] {
        self.compiler.messages()
    }

    pub fn get_names(&self) -> &[String] {
        self.compiler.names()
    }

    pub fn next_token(&mut self) -> Box<dyn Token> {
        while self.cur.cp().is_some() {
            while self.cur.is_whitespace() {
                self.cur.next();
            }

            let mut start = self.cur.clone();
            let mut end = self.cur.clone();

            if start.cp().is_none() {
                return Box::new(token::EOPToken::new(&start));
            }
```

```
let mut attr = "".to_string();

match start.cp().unwrap() {
    '<' => {
        end.next();
        if end.cp().is_none() {
            self.write_syntax_err(start);
            self.cur = end.clone();
            return Box::new(token::EOPToken::new(&end));
        }

        let c = end.cp().unwrap();
        match c {
            '=' => {
                end.next();
                self.cur = end.clone();
                return Box::new(token::OperationToken::new("<=", &start, &end));
            }
            _ => {
                while end.is_numeric() {
                    attr.push(end.cp().unwrap());
                    end.next();
                }

                if !attr.is_empty() && end.cp().unwrap().eq(&'>') {
                    end.next();
                    self.cur = end.clone();

                    return Box::new(token::NumberToken::new(
                        attr.parse::<i128>().unwrap(),
                        start,
                        end,
                    ));
                }

                self.write_syntax_err(start);
                self.skip_err();
            }
        }
    }
    '=' => {
        end.next();
        let mut result = OperationToken::new("=", &start, &end);

        if let Some('=') = end.cp() {
```

```
                        end.next();
                        result = OperationToken::new("==", &start, &end);
                    }

                    self.cur = end;
                    return Box::new(result);
                }

                _ => {
                    if end.is_alphanumeric() {
                        let mut last = 0;
                        let mut counter = 0;
                        let mut last_iter = None;
                        while end.is_alphanumeric() {
                            counter += 1;
                            attr.push(end.cp().unwrap());
                            if end.is_numeric() {
                                last = counter;
                                last_iter = Some(end.clone());
                            }

                            end.next();
                        }

                        if last > 0 {
                            let ident = attr[..last].to_string();
                            let code = self.compiler.add_name(&ident);

                            let mut last_iter = last_iter.unwrap();
                            last_iter.next();
                            self.cur = last_iter.clone();
                            return Box::new(IdentToken::new(code, start, last_iter));
                        } else {
                            self.write_syntax_err(start);
                            self.skip_err();
                        }
                    } else {
                        self.write_syntax_err(start);
                        self.skip_err();
                    }
                }
            }
        }

        Box::new(token::EOPToken::new(&self.cur))
    }
```

```
    fn write_syntax_err(&mut self, pos: Position) {
        if self.recovering {
            return;
        }

        self.compiler.add_message(true, "SYNTAX ERROR", pos);
    }

    fn skip_err(&mut self) {
        if self.recovering {
            self.cur.next();
            return;
        }

        self.recovering = true;

        let token = self.next_token();
        self.cur = token.get_coords().starting();

        self.recovering = false;
    }
}
```

Файл `main.rs`

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

use crate::domain::DomainTag;

pub mod compiler;
pub mod domain;
pub mod fragment;
pub mod message;
pub mod position;
pub mod scanner;
pub mod token;

fn main() {
    let args: Vec<String> = args().collect();

    if args.len() != 2 {
        println!("usage: cargo run <path_to_file>");
        return;
    }

    let file_path = &args[1];
```

```
    let tmp = fs::read_to_string(file_path).unwrap();
    let program = Box::leak(tmp.into_boxed_str());

    let compiler = compiler::Compiler::new();

    let mut scanner = scanner::Scanner::new(program, compiler);

    println!("TOKENS:");
    loop {
        let token = scanner.next_token();

        println!("{}", token);

        if let DomainTag::EOP = token.get_domain_tag() {
            break;
        }
    }

    println!("\nMESSAGES");

    let messages = scanner.get_messages();

    for (pos, msg) in messages {
        println!("{}: {}", msg, pos);
    }

    println!("\nIDENTIFIERS");

    let identifiers = scanner.get_names();
    for i in 0..identifiers.len() {
        println!("{}: {}", i, identifiers[i]);
    }
}
```

Тестирование

Входные данные

```
abc test123fff <1234>

ccc1c
<>

abc1   test123
```

```
<321
<1234>

<==>=  abc1

  <=
```

Вывод на `stdout`

```
TOKENS:
IDENT (2, 5)-(2, 12): 0
NUMBER (2, 16)-(2, 22): 1234
IDENT (4, 1)-(4, 5): 1
IDENT (7, 1)-(7, 5): 2
IDENT (7, 8)-(7, 15): 0
IDENT (9, 2)-(9, 5): 3
NUMBER (10, 1)-(10, 7): 1234
OPERATION (12, 1)-(12, 3): <=
OPERATION (12, 3)-(12, 4): =
OPERATION (12, 5)-(12, 6): =
IDENT (12, 8)-(12, 12): 2
OPERATION (14, 3)-(14, 5): <=
EOP (15, 1)-(15, 1)

MESSAGES
ERR: SYNTAX ERROR: (2, 1)
ERR: SYNTAX ERROR: (2, 12)
ERR: SYNTAX ERROR: (4, 5)
ERR: SYNTAX ERROR: (9, 1)
ERR: SYNTAX ERROR: (12, 4)

IDENTIFIERS
0: test123
1: ccc1
2: abc1
3: 321
```

Вывод

В ходе выполнения лабораторной работы 1.3 были получены навыки по реализации объектно-ориентированного лексического анализатора. В соответствии с индивидуальным вариантом были реализованы такие фазы стадии анализа как чтение входного потока и лексических анализ. Т.к. для выполнения лабораторной работы был выбран язык Rust, который не обладает “классическими” возможностями объектно ориентированных языков, приходилось активно

пользовается концепцией trait объектов.