一、 实验目的

1. 对使用SysY 语言书写的源代码进行词法分析

二、 实验内容与实验要求

二、实验内容

1.对使用SysY 语言书写的源代码进行词法分析；

2 编制测试程序；

3.调试程序

三、实验要求

1、编写一个程序，对使用SysY 语言书写的源代码进行词法分析，并打印分析结果；

2、程序要能够查出 SysY 源代码中可能包含的词法错误；

3、实验报告中需含有词法错误的详细实例；

4、以 WORD 附件形式上交程序源代码及实验报告。

三、 设计方案与算法描述

使用 rust 语言和 nom 框架进行编写

nom 是一个组合子框架，语法分析组合子 是一个高阶函数 ，它接受几个的语法分析器作为输入，并返回一个新的语法分析函器作为其输出。 在这个上下文中， 语法分析器 是一个函数，它接受字符串作为输入，返回的一些结构作为输出，通常为 分析树 或一组索引表示在字符串中成功停止分析的位置。 分析器组合子使用 递归下降分析 战略，提倡模块式建造和测试。 这种分析技术是所谓的 组合分析。

四、 测试结果

```
int a;
int b;


int main() {
    a = 10;
    b = 20;
    int c;
    c = a + b;
    return c;
}
```

```
❯ ./target/release/lab03-lexer ../sysy-sample/add.sy.c
tokenize file: `add.sy.c`
<primitive-type, int>
<ident, a>
<`;`>
<primitive-type, int>
<ident, b>
<`;`>
<primitive-type, int>
<ident, main>
<`(`>
<`)`>
<`{`>
<ident, a>
<`=`>
<int-lit, 10>
<`;`>
<ident, b>
<`=`>
<int-lit, 20>
<`;`>
<primitive-type, int>
<ident, c>
<`;`>
<ident, c>
<`=`>
<ident, a>
<`+`>
<ident, b>
<`;`>
<return>
<ident, c>
<`;`>
<`}`>
-----------------done-----------------
```

```
int g;

int h;

int f;

int e;

int eightwhile() {

  int a;

  a = 5;

  int b;

  int c;

  b = 6;

  c = 7;

  int d;
```

```
d = 10;
while (a < 20) {
    a = a + 3;
    while (b < 10) {
        b = b + 1;
        while (c == 7) {
            c = c - 1;
            while (d < 20) {
                d = d + 3;
                while (e > 1) {
                    e = e - 1;
                    while (f > 2) {
                        f = f - 2;
                        while (g < 3) {
                            g = g + 10;
                            while (h < 10) {
                                h = h + 8;
                            }
                            h = h - 1;
                        }
                        g = g - 8;
                    }
```

```
                f = f + 1;
            }
            e = e + 1;
        }
        d = d - 1;
    }
    c = c + 1;
}
b = b - 2;
}

return (a + (b + d) + c) - (e + d - g + h);
}

int main() {
    g = 1;
    h = 2;
    e = 4;
    f = 6;
    return eightwhile();
}
```

```
<`=`>
<ident, f>
<`+`>
<int-lit, 1>
<`;`>
<`}`>
<ident, e>
<`=`>
<ident, e>
<`+`>
<int-lit, 1>
<`;`>
<`}`>
<ident, d>
<`=`>
<ident, d>
<`-`>
<int-lit, 1>
<`;`>
<`}`>
<ident, c>
<`=`>
<ident, c>
<`+`>
<int-lit, 1>
<`;`>
<`}`>
<ident, b>
<`=`>
<ident, b>
<`-`>
<int-lit, 2>
<`;`>
<`}`>
<return>
<`(`>
<ident, a>
<`+`>
<`(`>
<ident, b>
<`+`>
<ident, d>
<`)`>
<`+`>
<ident, c>
<`)`>
<`-`>
<`(`>
<ident, e>
<`+`>
<ident, d>
<`-`>
<ident, g>
<`+`>
<ident, h>
<`)`>
<`;`>
<`}`>
<primitive-type, int>
<ident, main>
<`(`>
<`)`>
<`{`>
<ident, g>
<`=`>
<int-lit, 1>
<`;`>
<ident, h>
<`=`>
<int-lit, 2>
<`;`>
<ident, e>
<`=`>
<int-lit, 4>
<`;`>
<ident, f>
<`=`>
<int-lit, 6>
<`;`>
<return>
<ident, eightwhile>
<`(`>
<`)`>
<`;`>
<`}`>
-----------------done-----------------
```

int main(int argc, char *argv[]) { return 0; }

int a = 0xGGG;

```
❯ ./target/release/lab03-lexer bad-sample/illegal_digit.sy.c
tokenize file: `illegal_digit.sy.c`
<primitive-type, int>
<ident, main>
<`(`>
<primitive-type, int>
<ident, argc>
<`,`>
<primitive-type, char>
<`*`>
<ident, argv>
<`[`>
<`]`>
<`)`>
<`{`>
<return>
<int-lit, 0>
<`;`>
<`}`>
<primitive-type, int>
<ident, a>
<`=`>
parse_int_error

  × parse int literal GGG error, `invalid digit found in string`!
   ╭─[illegal_digit.sy.c:2:1]
 2 │
 3 │ int a = 0xGGG;
   ·          ▲
   ·          ╰── here
   ╰────

----------------done----------------
```

int a = 1;
#include <stdio.h>

```
❯ ./target/release/lab03-lexer bad-sample/illegal_token.sy.c
tokenize file: `illegal_token.sy.c`
<primitive-type, int>
<ident, a>
<`=`>
<int-lit, 1>
<`;`>
illegal_token

  × illegal token
   ╭─[illegal_token.sy.c:1:1]
 1 │ int a = 1;
 2 │ #include <stdio.h>
   · ▲
   · ╰── here
   ╰────
```

```
❯ ./target/release/lab03-lexer bad-sample/illegal_token.sy.c
tokenize file: `illegal_token.sy.c`
<primitive-type, int>
<ident, a>
<`=`>
<int-lit, 1>
<`;`>
illegal_token

  × illegal token
   ╭─[illegal_token.sy.c:1:1]
 1 │ int a = 1;
 2 │ #include <stdio.h>
   · ▲
   · ╰── here
   ╰────
```

int a =

    0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;

```
    0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;
❯ ./target/release/lab03-lexer bad-sample/too_big.sy.c
tokenize file: `too_big.sy.c`
<primitive-type, int>
<ident, a>
<`=`>
parse_int_error

  × parse int literal fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff error, `number too large to fit
  │ in target type`!
   ╭─[too_big.sy.c:1:1]
 1 │ int a =
 2 │     0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;
   ·     ▲
   ·     ╰── here
   ╰────

----------------done----------------
```

五、 源代码

use std::fmt::Display;

#[derive(Clone, Copy, PartialEq, Eq, Debug)]

pub enum PrimitiveType {

    Int,

    Float,

    Bool,

    Char,

    Void,

}

```
impl Display for PrimitiveType {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            PrimitiveType::Int => write!(f, "int"),
            PrimitiveType::Float => write!(f, "float"),
            PrimitiveType::Bool => write!(f, "bool"),
            PrimitiveType::Char => write!(f, "char"),
            PrimitiveType::Void => write!(f, "void"),
        }
    }
}

#[derive(Clone, PartialEq, Debug)]
pub enum Token {
    Illegal,
    Eof,
    // 字面量
    IntLit(i32),
    FloatLit(f32),
    StrLit(String),
    CharLit(u8), // ascii char only
```

```
// 关键字&标识符&基本类型
BoolLit(bool),
Null,
If,
Else,
While,
Break,
Continue,
Return,
Extern,                             //声明外部函数
Const,                              //声明编译时常量
PrimitiveType(PrimitiveType), //基本类型
Ident(String),
// 算符
Equal, // 声明，赋值
// 算术运算符
Plus,     //加法
Minus,    //减法/负号
Star,     //乘法，解引用，声明指针类型
Slash,    //除法
Percent, //取余
// 逻辑运算符
```

AmpersandAmpersand, //逻辑与

PipePipe, //逻辑或

Bang, //逻辑非

//关系运算符

EqualEqual, //判等

BangEqual, //判不等

Less, //小于

Greater, //大于

LessEqual, //小于等于

GreaterEqual, //大于等于

// 位运算符

Tilde, //按位取反

Caret, //按位异或

Ampersand, //按位与，取引用

Pipe, //按位或

// 标点符号

Comma, //, 分割数组元素/函数参数/对象属性

SemiColon, //; 语句末尾结束符, for 分割

LParen, //(函数调用/函数参数参数/表达式分组

RParen, //) 函数调用/函数参数/表达式分组

LBrack, //[数组索引/数组声明

RBrack, //] 数组索引/数组声明

```
    LBrace,       //{  代码块/函数体/条件分支体/循环体/数组构造
    RBrace,       //}  代码块/函数体/条件分支体/循环体/数组构造
    DotDotDot, // `...`  可变参数
}
impl Display for Token {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Token::Illegal => write!(f, "<illegal>"),
            Token::Eof => write!(f, "<eof>"),
            Token::PrimitiveType(type_) => write!(f, "<primitive-type, {}>",
type_),
            Token::IntLit(lit) => write!(f, "<int-lit, {}>", lit),
            Token::FloatLit(lit) => write!(f, "<float-lit, {}>", lit),
            Token::StrLit(lit) => write!(f, "<str-lit, \"{}\">", lit.escape_default()),
            Token::BoolLit(lit) => write!(f, "<bool-lit, {}>", lit),
            Token::CharLit(lit) => write!(f, "<char-lit, '{}'>", (*lit as
char).escape_default()),
            Token::Null => write!(f, "<null>"),
            Token::If => write!(f, "<if>"),
            Token::Else => write!(f, "<else>"),
            Token::While => write!(f, "<while>"),
            Token::Break => write!(f, "<break>"),
```

```
Token::Continue => write!(f, "<continue>"),

Token::Return => write!(f, "<return>"),

Token::Extern => write!(f, "<extern>"),

Token::Const => write!(f, "<const>"),

Token::Ident(ident) => write!(f, "<ident, {}>", ident),

Token::Equal => write!(f, "<`=`>"),

Token::Plus => write!(f, "<`+`>"),

Token::Minus => write!(f, "<`-`>"),

Token::Star => write!(f, "<`*`>"),

Token::Slash => write!(f, "<`/`>"),

Token::Percent => write!(f, "<`%`>"),

Token::AmpersandAmpersand => write!(f, "<`&&`>"),

Token::PipePipe => write!(f, "<`||`>"),

Token::Bang => write!(f, "<`!`>"),

Token::EqualEqual => write!(f, "<`==`>"),

Token::BangEqual => write!(f, "<`!=`>"),

Token::Less => write!(f, "<`<`>"),

Token::Greater => write!(f, "<`>`>"),

Token::LessEqual => write!(f, "<`<=`>"),

Token::GreaterEqual => write!(f, "<`>=`>"),

Token::Tilde => write!(f, "<`~`>"),

Token::Caret => write!(f, "<`^`>"),
```

```
            Token::Ampersand => write!(f, "<`&`>"),
            Token::Pipe => write!(f, "<`|`>"),
            Token::Comma => write!(f, "<`,`>"),
            Token::SemiColon => write!(f, "<`;`>"),
            Token::LParen => write!(f, "<`(`>"),
            Token::RParen => write!(f, "<`)`>"),
            Token::LBrack => write!(f, "<`[`>"),
            Token::RBrack => write!(f, "<`]`>"),
            Token::LBrace => write!(f, "<`{{`>"),
            Token::RBrace => write!(f, "<`}}`>"),
            Token::DotDotDot => write!(f, "<`...`>"),
        }
    }
}

static RESERVED: phf::Map<&'static str, Token> = phf::phf_map! {
    "null" => Token::Null,
    "if" => Token::If,
    "else" => Token::Else,
    "while" => Token::While,
    "break" => Token::Break,
    "continue" => Token::Continue,
```

```
    "return" => Token::Return,
    "extern" => Token::Extern,
    "const" => Token::Const,
    "int" => Token::PrimitiveType(PrimitiveType::Int),
    "float" => Token::PrimitiveType(PrimitiveType::Float),
    "bool" => Token::PrimitiveType(PrimitiveType::Bool),
    "char" => Token::PrimitiveType(PrimitiveType::Char),
    "void" => Token::PrimitiveType(PrimitiveType::Void),
    "true" => Token::BoolLit(true),
    "false" => Token::BoolLit(false),
};


pub fn ident_or_reserved(ident: &str) -> Token {
    RESERVED
        .get(ident)
        .cloned()
        .unwrap_or(Token::Ident(ident.to_owned()))
}
use std::num::{ParseFloatError, ParseIntError};


use crate::{span::Span, tokens::Token};
use nom::{
```

```
    error::{ContextError, FromExternalError, ParseError},
    IResult,
};
use thiserror::Error;


#[derive(Error, Debug, Clone)]
pub enum LexError<'a> {
    #[error("eof")]
    Eof,
    #[error("unknown error")]
    Unknown,
    #[error("illegal token")]
    IllegalToken,
    #[error("expected one of {1:?}, found `{0}`!")]
    UnexpectedChar(char, Vec<char>),
    #[error("parse float literal {1} error, `{0}`!")]
    ParseFloatError(ParseFloatError, &'a str),
    #[error("parse int literal {1} error, `{0}`!")]
    ParseIntError(ParseIntError, &'a str),
    #[error("EscapeCharError: escape char `{0}` is not supported!")]
    EscapeCharError(char),
}
```

```
impl<'a> LexError<'a> {
    pub fn code(&self) -> &'static str {
        match self {
            LexError::Eof => "eof",
            LexError::Unknown => "unknown",
            LexError::UnexpectedChar(_, _) => "unexpected_char",
            LexError::ParseFloatError(_, _) => "parse_float_error",
            LexError::ParseIntError(_, _) => "parse_int_error",
            LexError::EscapeCharError(_) => "escape_char_error",
            LexError::IllegalToken => "illegal_token",
        }
    }
}

#[derive(Debug)]
pub struct SourcedLexError<'a> {
    pub error: LexError<'a>,
    pub span: Span<'a>,
}

impl<'a> SourcedLexError<'a> {
```

```
    pub fn is_eof(&self) -> bool {
        if let LexError::Eof = self.error {
            true
        } else {
            false
        }
    }

    pub fn span(&self) -> miette::SourceSpan {
        self.span.location_offset().into()
    }
}

impl<'a> ParseError<Span<'a>> for SourcedLexError<'a> {
    fn from_error_kind(input: Span<'a>, _: nom::error::ErrorKind) -> Self {
        SourcedLexError {
            error: LexError::Unknown,
            span: input,
        }
    }

    fn append(input: Span<'a>, _: nom::error::ErrorKind, _: Self) -> Self {
```

```
        SourcedLexError {
            error: LexError::Unknown,
            span: input,
        }
    }

    fn from_char(input: Span<'a>, _: char) -> Self {
        if let Some(ch) = input.fragment().chars().next() {
            SourcedLexError {
                error: LexError::UnexpectedChar(ch, vec![]),
                span: input,
            }
        } else {
            SourcedLexError {
                error: LexError::Eof,
                span: input,
            }
        }
    }

    fn or(mut self, mut other: Self) -> Self {
        match (&mut self.error, &mut other.error) {
```

```
            (LexError::UnexpectedChar(_, expected_a), LexError::UnexpectedChar(_, expected_b)) => {
                expected_a.append(expected_b);
                self
            }
            (LexError::UnexpectedChar(_, _), _) => self,
            _ => other,
        }
    }
}

impl ContextError<Span<'_>> for SourcedLexError<'_> {
    fn add_context(_input: Span<'_>, _ctx: &'static str, other: Self) -> Self {
        other
    }
}

pub type LexResult<'a, T = Token> = IResult<Span<'a>, T, SourcedLexError<'a>>;

impl<'a> FromExternalError<Span<'a>, LexError<'a>> for SourcedLexError<'a> {
    fn from_external_error(input: Span<'a>, _: nom::error::ErrorKind, e: LexError<'a>) -> Self {
```

```rust
            SourcedLexError {
                error: e,
                span: input,
            }
        }
}
use crate::error::{LexError, LexResult, SourcedLexError};
use crate::span::{Meta, Span};
use crate::tokens::{ident_or_reserved, Token};
use miette::{miette, LabeledSpan, NamedSource, Severity};
use nom::branch::alt;
use nom::bytes::complete::{tag, tag_no_case, take_until};
use nom::character::complete::{alpha1, alphanumeric1, char, digit1, multispace0,
one_of};
use nom::combinator::{iterator, map, opt, recognize};
use nom::multi::many0_count;
use nom::sequence::{delimited, pair, tuple};
use nom::Err;

pub fn lex(filename: &str, input: &str) {
    let span = Span::new_extra(input, Meta::new(filename));
    let mut it = iterator(span, lex_token);
```

```
it.for_each(|token| println!("{token}"));
match it.finish().err() {
    Some(Err::Failure(err)) => {
        if err.is_eof() {
            return;
        }
        let code = err.error.code();
        let msg = err.error.to_string();
        let labels = vec![LabeledSpan::at(err.span(), "here")];
        let report = miette!(
            severity = Severity::Error,
            code = code,
            labels = labels,
            "{}",
            msg
        )
        .with_source_code(NamedSource::new(filename, input.to_string()));
        println!("{:?}", report);
    }
    Some(Err::Error(_)) => {
        unreachable!()
    }
```

```
            Some(Err::Incomplete(_)) => {
                unreachable!()
            }
            None => {}
        }
    }

fn raise_failure<'a>(err: LexError<'a>) -> impl FnMut(Span<'a>) -> LexResult<'a,
Token> {
    move |input: Span| {
        LexResult::Err(nom::Err::Failure(SourcedLexError {
            error: err.clone(),
            span: input,
        }))
    }
}

fn eof(input: Span) -> LexResult {
    if input.is_empty() {
        LexResult::Err(nom::Err::Failure(SourcedLexError {
            error: LexError::Eof,
            span: input,
```

```
        }))
    } else {
        LexResult::Err(nom::Err::Error(SourcedLexError {
            error: LexError::Unknown,
            span: input,
        }))
    }
}


macro_rules! syntax {
    ($func_name: ident, $tag_string: literal, $output_token: expr) => {
        fn $func_name(input: Span) -> LexResult<Token> {
            map(tag($tag_string), |_| $output_token)(input)
        }
    };
}

pub fn lex_token(input: Span) -> LexResult<Token> {
    delimited(
        skip,
        alt((
```

```
            lex_lit,
            lex_operators,
            lex_punctuator,
            lex_ident_or_reserved,
            eof,
            raise_failure(LexError::IllegalToken),
        )),
        skip,
    )(input)
}

fn skip(input: Span) -> LexResult<()> {
    let multi_line_comment = tuple((
        multispace0,
        tag("/*"),
        take_until("*/"),
        tag("*/"),
        multispace0,
    ));
    let one_line_comment = tuple((
        multispace0,
        tag("//"),
```

```
            take_until("\n"),
            tag("\n"),
            multispace0,
        ));
        let commets = recognize(many0_count(tuple((
            alt((one_line_comment, multi_line_comment)),
            multispace0,
        ))));

        return map(tuple((multispace0, opt(commets))), |_| ())(input);
}

syntax!(lex_equal, "=", Token::Equal);
syntax!(lex_plus, "+", Token::Plus);
syntax!(lex_minus, "-", Token::Minus);
syntax!(lex_star, "*", Token::Star);
syntax!(lex_slash, "/", Token::Slash);
syntax!(lex_percent, "%", Token::Percent);
syntax!(lex_ampersand_ampersand, "&&", Token::AmpersandAmpersand);
syntax!(lex_pipe_pipe, "||", Token::PipePipe);
syntax!(lex_bang, "!", Token::Bang);
syntax!(lex_equal_equal, "==", Token::EqualEqual);
```

```
syntax!(lex_bang_equal, "!=", Token::BangEqual);

syntax!(lex_less, "<", Token::Less);

syntax!(lex_greater, ">", Token::Greater);

syntax!(lex_less_equal, "<=", Token::LessEqual);

syntax!(lex_greater_equal, ">=", Token::GreaterEqual);

syntax!(lex_tilde, "~", Token::Tilde);

syntax!(lex_caret, "^", Token::Caret);

syntax!(lex_ampersand, "&", Token::Ampersand);

syntax!(lex_pipe, "|", Token::Pipe);

syntax!(lex_comma, ",", Token::Comma);

syntax!(lex_semi_colon, ";", Token::SemiColon);

syntax!(lex_lparen, "(", Token::LParen);

syntax!(lex_rparen, ")", Token::RParen);

syntax!(lex_lbrack, "[", Token::LBrack);

syntax!(lex_rbrack, "]", Token::RBrack);

syntax!(lex_lbrace, "{", Token::LBrace);

syntax!(lex_rbrace, "}", Token::RBrace);

syntax!(lex_dot_dot_dot, "...", Token::DotDotDot);

fn lex_operators(input: Span) -> LexResult<Token> {

    alt((

        lex_equal_equal,

        lex_equal,
```

```
        lex_plus,
        lex_minus,
        lex_star,
        lex_slash,
        lex_percent,
        lex_ampersand_ampersand,
        lex_pipe_pipe,
        lex_bang_equal,
        lex_bang,
        lex_less_equal,
        lex_less,
        lex_greater_equal,
        lex_greater,
        lex_tilde,
        lex_caret,
        lex_ampersand,
        lex_pipe,
    ))(input)
}

fn lex_punctuator(input: Span) -> LexResult<Token> {
    alt((
```

```
        lex_comma,
        lex_semi_colon,
        lex_lparen,
        lex_rparen,
        lex_lbrack,
        lex_rbrack,
        lex_lbrace,
        lex_rbrace,
        lex_dot_dot_dot,
    ))(input)
}

fn lex_ident_or_reserved(input: Span) -> LexResult<Token> {
    let (leftover, ident) = recognize(pair(
        alt((tag("_"), alpha1)),
        many0_count(alt((tag("_"), alphanumeric1))),
    ))(input)?;
    return Ok((leftover, ident_or_reserved(ident.fragment())));
}

use lex_char::lex as lex_char_lit;
use lex_float::lex as lex_float_lit;
```

```
use lex_int::lex as lex_int_lit;
use lex_str::lex as lex_str_lit;


fn lex_lit(input: Span) -> LexResult<Token> {
    alt((lex_str_lit, lex_char_lit, lex_float_lit, lex_int_lit))(input)
}


fn lex_operators(input: Span) -> LexResult<Token> {
    alt((
        lex_equal_equal,
        lex_equal,
        lex_plus,
        lex_minus,
        lex_star,
        lex_slash,
        lex_percent,
        lex_ampersand_ampersand,
        lex_pipe_pipe,
        lex_bang_equal,
        lex_bang,
        lex_less_equal,
        lex_less,
```

```
        lex_greater_equal,
        lex_greater,
        lex_tilde,
        lex_caret,
        lex_ampersand,
        lex_pipe,
    ))(input)
}

fn lex_punctuator(input: Span) -> LexResult<Token> {
    alt((
        lex_comma,
        lex_semi_colon,
        lex_lparen,
        lex_rparen,
        lex_lbrack,
        lex_rbrack,
        lex_lbrace,
        lex_rbrace,
        lex_dot_dot_dot,
    ))(input)
}
```

```
fn lex_ident_or_reserved(input: Span) -> LexResult<Token> {
    let (leftover, ident) = recognize(pair(
        alt((tag("_"), alpha1)),
        many0_count(alt((tag("_"), alphanumeric1))),
    ))(input)?;
    return Ok((leftover, ident_or_reserved(ident.fragment())));
}

use lex_char::lex as lex_char_lit;
use lex_float::lex as lex_float_lit;
use lex_int::lex as lex_int_lit;
use lex_str::lex as lex_str_lit;

fn lex_lit(input: Span) -> LexResult<Token> {
    alt((lex_str_lit, lex_char_lit, lex_float_lit, lex_int_lit))(input)
}

mod lex_str {
    use nom::{
        branch::alt,
        bytes::complete::{is_not, take},
```

```
    character::complete::char,
    combinator::{cut, map, map_res, value, verify},
    multi::fold_many0,
    sequence::{delimited, preceded},
    Parser,
};

use crate::error::LexError;

use super::{LexResult, Span, Token};

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum StringFragment<'a> {
    Literal(&'a str),
    EscapedChar(char),
}

fn lex_escaped(input: Span) -> LexResult<char> {
    preceded(
        char('\\'),
        cut(alt((
            value('\t', char('t')),
```

```
                value('\r', char('r')),
                value('\n', char('n')),
                value('"', char('"')),
                value('\\', char('\\')),
                map_res(take(1usize), |input: Span| {
                    Err(LexError::EscapeCharError(
                        input.fragment().chars().next().unwrap(),
                    ))
                }),
            ))),
        )(input)
    }

    fn lex_lit<'a>(input: Span<'a>) -> LexResult<&'a str> {
        let not_quote_slash = is_not("\"\\");
        let (leftover, parsed) =
            verify(not_quote_slash,                                    |s:
&Span| !s.fragment().is_empty()).parse(input)?;
        Ok((leftover, parsed.fragment()))
    }

    fn lex_fragment(input: Span) -> LexResult<StringFragment> {
```

```
        alt((
            map(lex_escaped, StringFragment::EscapedChar),
            map(lex_lit, StringFragment::Literal),
        ))(input)
    }

    pub fn lex(input: Span) -> LexResult<Token> {
        let parse_inner = fold_many0(lex_fragment, String::new, |mut acc, fragment| {
            match fragment {
                StringFragment::Literal(s) => acc.push_str(s),
                StringFragment::EscapedChar(c) => acc.push(c),
            }
            acc
        });
        map(delimited(char(""), parse_inner, char("")), Token::StrLit)(input)
    }

    #[cfg(test)]
    mod test {
        use crate::span::Meta;
        use crate::tokens::Token;
```

```
use super::lex;
use super::Span;
#[test]
fn test_str() {
    assert_eq!(
        lex(Span::new_extra(r###""hello world""###, Meta::new("")))
            .unwrap()
            .1,
        Token::StrLit("hello world".to_owned())
    );
    assert_eq!(
        lex(Span::new_extra(r###""hello\nworld""###, Meta::new("")))
            .unwrap()
            .1,
        Token::StrLit("hello\nworld".to_owned())
    );
    assert_eq!(
        lex(Span::new_extra(r###""hello\tworld""###, Meta::new("")))
            .unwrap()
            .1,
        Token::StrLit("hello\tworld".to_owned())
```

```
            );
        }
    }
}

mod lex_char {
    use nom::branch::alt;
    use nom::bytes::complete::take;
    use nom::character::complete::char;
    use nom::combinator::cut;
    use nom::combinator::map;
    use nom::combinator::map_res;
    use nom::combinator::value;
    use nom::sequence::delimited;
    use nom::sequence::preceded;
    use nom::Parser;

    use crate::error::LexError;
    use crate::tokens::Token;

    use super::LexResult;
    use super::Span;
```

```
fn lex_escaped(input: Span) -> LexResult<u8> {
    preceded(
        char('\\'),
        cut(alt((
            value(b'\t', char('t')),
            value(b'\r', char('r')),
            value(b'\n', char('n')),
            value(b'\", char('\")),
            value(b'\\', char('\\')),
            map_res(take(1usize), |input: Span| {

Err(LexError::EscapeCharError(input.chars().next().unwrap()))
            }),
        ))),
    )(input)
}


fn lex_normal(input: Span) -> LexResult<u8> {
    let (leftover, parsed) = nom::bytes::complete::take(1usize).parse(input)?;
    let ch = parsed.fragment().chars().next().unwrap();
    assert!(ch.is_ascii(), "char literal must be ascii");
    Ok((leftover, ch as u8))
```

```
}

pub fn lex(input: Span) -> LexResult<Token> {
    map(
        delimited(char('\''), alt((lex_escaped, lex_normal)), char('\'')),
        Token::CharLit,
    )(input)
}

#[cfg(test)]
mod test {
    use crate::span::Meta;
    use crate::tokens::Token;

    use super::lex;
    use super::Span;
    #[test]
    fn test_lex() {
        assert_eq!(
            lex(Span::new_extra("'a'", Meta::new(""))).unwrap().1,
            Token::CharLit(b'a')
        );
```

```
        assert_eq!(
            lex(Span::new_extra("'\\n'", Meta::new(""))).unwrap().1,
            Token::CharLit(b'\n')
        );
        assert_eq!(
            lex(Span::new_extra("'\\t'", Meta::new(""))).unwrap().1,
            Token::CharLit(b'\t')
        );
        assert_eq!(
            lex(Span::new_extra("'\\r'", Meta::new(""))).unwrap().1,
            Token::CharLit(b'\r')
        );
        assert_eq!(
            lex(Span::new_extra("'\\"', Meta::new(""))).unwrap().1,
            Token::CharLit(b'\")
        );
        assert_eq!(
            lex(Span::new_extra("'\\\\'", Meta::new(""))).unwrap().1,
            Token::CharLit(b'\\')
        );
    }
}
```

```
}

mod lex_float {
    use nom::Err;

    use crate::error::{LexError, SourcedLexError};

    use super::*;
    fn interger_fractional_expponent(input: Span) -> LexResult<Span> {
        let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
        let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);
        let exponent_part = pair(
            tag_no_case("e"),
            pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
        );
        recognize(tuple((integer_part, fractional_part, exponent_part)))(input)
    }

    fn interger_fractional(input: Span) -> LexResult<Span> {
        let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
        let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);
        recognize(tuple((integer_part, fractional_part)))(input)
```

```
}

fn interger_exponent(input: Span) -> LexResult<Span> {
    let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
    let exponent_part = pair(
        tag_no_case("e"),
        pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
    );
    recognize(tuple((integer_part, exponent_part)))(input)
}

fn no_interger_part(input: Span) -> LexResult<Span> {
    let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);
    let exponent_part = pair(
        tag_no_case("e"),
        pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
    );
    recognize(tuple((fractional_part, opt(exponent_part))))(input)
}

pub fn lex(input: Span) -> LexResult<Token> {
    let (leftover, parsed) = alt((
```

```
        interger_fractional_expponent,
        interger_exponent,
        interger_fractional,
        no_interger_part,
    ))(input.clone())?;
    let parsed = parsed.fragment();
    let ret = parsed.parse::<f32>().map_err(|err| {
        Err::Failure(SourcedLexError {
            error: LexError::ParseFloatError(err, *parsed),
            span: input,
        })
    })?;
    Ok((leftover, Token::FloatLit(ret)))
}

#[cfg(test)]
mod test {
    use crate::span::Meta;

    use super::lex;
    use super::Span;
    use super::Token;
```

```
#[test]
fn test_float() {
    assert_eq!(
        lex(Span::new_extra("1.0", Meta::new(""))).unwrap().1,
        Token::FloatLit(1.0)
    );
    assert_eq!(
        lex(Span::new_extra("1.0e1", Meta::new(""))).unwrap().1,
        Token::FloatLit(1.0e1)
    );
    assert_eq!(
        lex(Span::new_extra("1.0e-1", Meta::new(""))).unwrap().1,
        Token::FloatLit(1.0e-1)
    );
    assert_eq!(
        lex(Span::new_extra("1.0e+1", Meta::new(""))).unwrap().1,
        Token::FloatLit(1.0e+1)
    );
    assert_eq!(
        lex(Span::new_extra("1.0E1", Meta::new(""))).unwrap().1,
        Token::FloatLit(1.0E1)
    );
```

```
            assert_eq!(
                lex(Span::new_extra("1.0E-1", Meta::new(""))).unwrap().1,
                Token::FloatLit(1.0E-1)
            );
            assert_eq!(
                lex(Span::new_extra(".1E+1", Meta::new(""))).unwrap().1,
                Token::FloatLit(0.1E+1)
            );
        }
    }
}

mod lex_float {
    use nom::Err;

    use crate::error::{LexError, SourcedLexError};

    use super::*;
    fn interger_fractional_expponent(input: Span) -> LexResult<Span> {
        let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
        let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);
        let exponent_part = pair(
```

```
        tag_no_case("e"),
        pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
    );
    recognize(tuple((integer_part, fractional_part, exponent_part)))(input)
}

fn interger_fractional(input: Span) -> LexResult<Span> {
    let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
    let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);
    recognize(tuple((integer_part, fractional_part)))(input)
}

fn interger_exponent(input: Span) -> LexResult<Span> {
    let integer_part = pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1);
    let exponent_part = pair(
        tag_no_case("e"),
        pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
    );
    recognize(tuple((integer_part, exponent_part)))(input)
}

fn no_interger_part(input: Span) -> LexResult<Span> {
```

```
    )

    let fractional_part = pair(char::<_, SourcedLexError>('.'), digit1);

    let exponent_part = pair(
        tag_no_case("e"),
        pair(opt(one_of::<_, _, SourcedLexError>("+-")), digit1),
    );

    recognize(tuple((fractional_part, opt(exponent_part))))(input)
}

pub fn lex(input: Span) -> LexResult<Token> {
    let (leftover, parsed) = alt((
        interger_fractional_expponent,
        interger_exponent,
        interger_fractional,
        no_interger_part,
    ))(input.clone())?;
    let parsed = parsed.fragment();
    let ret = parsed.parse::<f32>().map_err(|err| {
        Err::Failure(SourcedLexError {
            error: LexError::ParseFloatError(err, *parsed),
            span: input,
        })
    })?;
```

```
    Ok((leftover, Token::FloatLit(ret)))
}

#[cfg(test)]
mod test {
    use crate::span::Meta;

    use super::lex;
    use super::Span;
    use super::Token;
    #[test]
    fn test_float() {
        assert_eq!(
            lex(Span::new_extra("1.0", Meta::new(""))).unwrap().1,
            Token::FloatLit(1.0)
        );
        assert_eq!(
            lex(Span::new_extra("1.0e1", Meta::new(""))).unwrap().1,
            Token::FloatLit(1.0e1)
        );
        assert_eq!(
            lex(Span::new_extra("1.0e-1", Meta::new(""))).unwrap().1,
```

```
            Token::FloatLit(1.0e-1)
        );
        assert_eq!(
            lex(Span::new_extra("1.0e+1", Meta::new(""))).unwrap().1,
            Token::FloatLit(1.0e+1)
        );
        assert_eq!(
            lex(Span::new_extra("1.0E1", Meta::new(""))).unwrap().1,
            Token::FloatLit(1.0E1)
        );
        assert_eq!(
            lex(Span::new_extra("1.0E-1", Meta::new(""))).unwrap().1,
            Token::FloatLit(1.0E-1)
        );
        assert_eq!(
            lex(Span::new_extra(".1E+1", Meta::new(""))).unwrap().1,
            Token::FloatLit(0.1E+1)
        );
    }
  }
}
```

```
mod lex_int {
    use crate::error::{LexError, SourcedLexError};

    use super::*;
    use nom::{character::complete::i32, sequence::preceded, Err};

    fn hex(input: Span) -> LexResult<i32> {
        let (leftover, parsed) = preceded(tag_no_case("0x"),
alphanumeric1)(input.clone())?;
        let parsed = parsed.fragment();
        let ret = i32::from_str_radix(parsed, 16).map_err(|err| {
            Err::Failure(SourcedLexError {
                error: LexError::ParseIntError(err, *parsed),
                span: input,
            })
        })?;
        Ok((leftover, ret))
    }

    fn bin(input: Span) -> LexResult<i32> {
        let (leftover, parsed) = preceded(tag_no_case("0b"),
alphanumeric1)(input.clone())?;
```

```
        let parsed = parsed.fragment();
        let ret = i32::from_str_radix(parsed, 2).map_err(|err| {
            Err::Failure(SourcedLexError {
                error: LexError::ParseIntError(err, *parsed),
                span: input,
            })
        })?;
        Ok((leftover, ret))
    }

    fn oct(input: Span) -> LexResult<i32> {
        let (leftover, parsed) = preceded(tag_no_case("0o"), alphanumeric1)(input.clone())?;
        let parsed = parsed.fragment();
        let ret = i32::from_str_radix(parsed, 8).map_err(|err| {
            Err::Failure(SourcedLexError {
                error: LexError::ParseIntError(err, *parsed),
                span: input,
            })
        })?;
        Ok((leftover, ret))
    }
```

```
fn dec(input: Span) -> LexResult<i32> {
    i32(input)
}

pub fn lex(input: Span) -> LexResult<Token> {
    map(alt((hex, bin, oct, dec)), Token::IntLit)(input)
}

#[cfg(test)]
mod test {
    use crate::span::Meta;

    use super::super::Span;
    use super::lex;
    use super::Token;

    #[test]
    fn test_interger() {
        assert_eq!(
            lex(Span::new_extra("0x1", Meta::new(""))).unwrap().1,
            Token::IntLit(1)
```

```
        );
        assert_eq!(
            lex(Span::new_extra("0b1", Meta::new(""))).unwrap().1,
            Token::IntLit(1)
        );
        assert_eq!(
            lex(Span::new_extra("0o1", Meta::new(""))).unwrap().1,
            Token::IntLit(1)
        );
        assert_eq!(
            lex(Span::new_extra("1", Meta::new(""))).unwrap().1,
            Token::IntLit(1)
        );
    }
  }
}
#[derive(Debug, Clone)]
pub struct Meta<'a> {
    pub filename: &'a str,
}

impl<'a> Meta<'a> {
```

```rust
    pub fn new(filename: &'a str) -> Self {
        Self { filename }
    }
}

pub type Span<'a> = nom_locate::LocatedSpan<&'a str, Meta<'a>>;
use std::{
    env,
    fs::File,
    io::{read_to_string, Result},
    path::PathBuf,
};

use lex::lex;
pub mod error;
pub mod lex;
pub mod span;
pub mod tokens;

extern crate nom;
extern crate nom_locate;
```

```
fn main() -> Result<()> {
    for path in env::args().skip(1).map(|s| PathBuf::from(s)) {
        let file_name = path.file_name().unwrap().to_str().unwrap();
        println!("tokenize file: `{file_name}`");
        let code = read_to_string(File::open(&path)?)?;
        lex(file_name, &code);
        println!("-----------------done-----------------");
    }
    Ok(())
}
```