Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЕТ

к лабораторной работе №3

на тему

**СИНТАКСИЧЕСКИЙ АНАЛИЗ**

Студент А. В. Скворцов

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc159418165)

[2 Краткие теоретические сведения 4](#_Toc159418166)

[3 Результаты выполнения лабораторной работы 5](#_Toc159418167)

[Заключение 7](#_Toc159418168)

[Список использованных источников 8](#_Toc159418169)

[Приложение А (обязательное) Листинг кода 9](#_Toc159418170)

1. **ЦЕЛЬ РАБОТЫ**

Целью выполнения данной лабораторной работы являются освоение работы с существующими синтаксическими анализаторами, разработка синтаксического анализатора подмножества языка программирования, определенного в лабораторной работе 1, определение синтаксических правил, построение дерева разбора, выявление синтаксических ошибок.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Разработанный программный продукт представляет собой синтаксический анализатор для языка программирования Python. Он разбивает код на так называемые вершины абстрактного синтаксического дерева.

Синтаксический анализ – это процесс, который определяет, принадлежит ли некоторая последовательность лексем языку, порождаемому грамматикой. Обычно представляется в виде дерева для удобства понимания.

Метод рекурсивного спуска (англ. Recursive descent parser) — алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ. Применения правил последовательно, слева направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов синтаксического анализа, подходящий для полностью ручной реализации.

Синтаксический LL-анализатор (LL parser) — в информатике нисходящий синтаксический анализатор для некоторого подмножества контекстно-свободных грамматик, известных как LL-грамматики. При этом не все контекстно-свободные грамматики являются LL-грамматиками.

Буквы L в выражении «LL-анализатор» означают, что входная строка анализируется слева направо (left to right), и при этом строится её левосторонний вывод (leftmost derivation).

LL-анализатор называется LL(k)-анализатором, если данный анализатор использует предпросмотр на k токенов (лексем) при разборе входного потока. Грамматика, которая может быть распознана LL(k)-анализатором без возвратов к предыдущим символам, называется LL(k)-грамматикой. Язык, который может быть представлен в виде LL(k)-грамматики, называется LL(k)o-языком. Существуют LL(k+n)-языки, которые не являются LL(k)-языками [1].

1. **РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**

В рамках лабораторной работы был разработан синтаксический анализатор языка Python, для тестирования был создан файл, который содержит основные конструкции языка для наглядного показа построения абстрактного синтаксического дерева (рисунок 1).

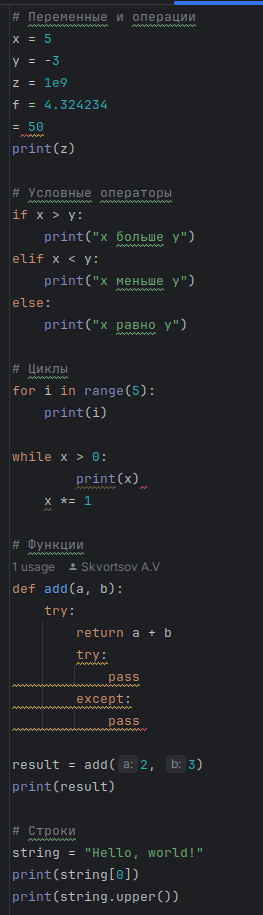


Рисунок 1 – Тестовый код

После того, как на этот код был запущен синтаксический анализатор, можно увидеть результаты выполнения программы.

На рисунке ниже приведен вывод абстрактного синтаксического дерева на примере разбора условного оператора if (рисунок 2).

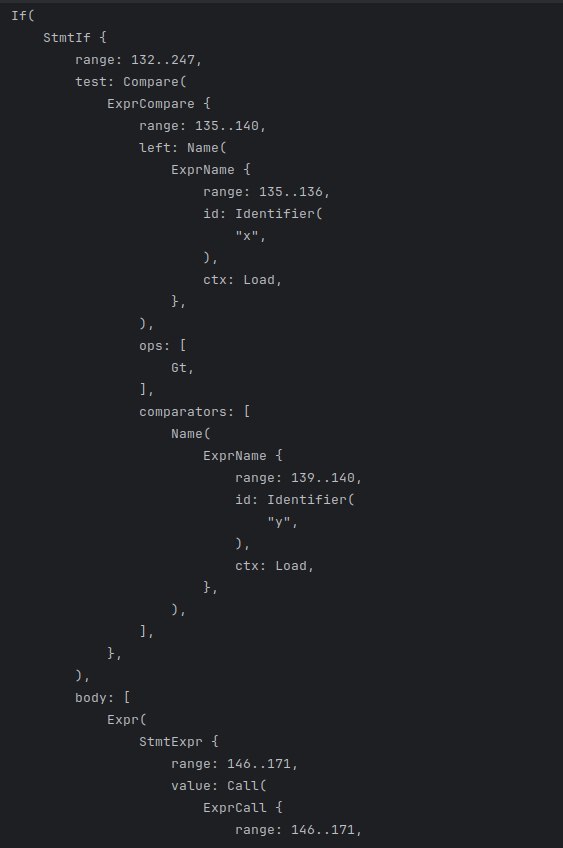


Рисунок 2 – AST дерево

На следующем рисунке приведен вывод программы при ошибке отступа в языке Python (рисунок 3).

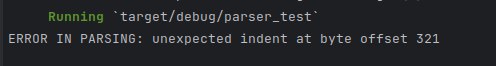


Рисунок 3 – Ошибка отступа

## 

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы был разработан синтаксический анализатор подмножества языка программирования Python, который в результате своей работы получает массив определенных выражений либо ошибку анализа. Был продемонстрирован результат работы программы при обнаружении ошибок.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Синтаксический\_анализатор – Дата доступа: 22.02.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – ast.rs

//! `builtin\_types` in asdl.py and Attributed

use crate::bigint::BigInt;

pub type String = std::string::String;

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

pub struct Identifier(String);

impl Identifier {

#[inline]

pub fn new(s: impl Into<String>) -> Self {

Self(s.into())

}

}

impl Identifier {

#[inline]

pub fn as\_str(&self) -> &str {

self.0.as\_str()

}

}

impl std::cmp::PartialEq<str> for Identifier {

#[inline]

fn eq(&self, other: &str) -> bool {

self.0 == other

}

}

impl std::cmp::PartialEq<String> for Identifier {

#[inline]

fn eq(&self, other: &String) -> bool {

&self.0 == other

}

}

impl std::ops::Deref for Identifier {

type Target = str;

#[inline]

fn deref(&self) -> &Self::Target {

self.0.as\_str()

}

}

impl AsRef<str> for Identifier {

#[inline]

fn as\_ref(&self) -> &str {

self.0.as\_str()

}

}

impl AsRef<String> for Identifier {

#[inline]

fn as\_ref(&self) -> &String {

&self.0

}

}

impl std::fmt::Display for Identifier {

fn fmt(&self, f: &mut std::fmt::Formatter<'\_>) -> std::fmt::Result {

self.0.fmt(f)

}

}

impl From<Identifier> for String {

#[inline]

fn from(id: Identifier) -> String {

id.0

}

}

impl From<String> for Identifier {

#[inline]

fn from(id: String) -> Self {

Self(id)

}

}

impl<'a> From<&'a str> for Identifier {

#[inline]

fn from(id: &'a str) -> Identifier {

id.to\_owned().into()

}

}

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

pub struct Int(u32);

impl Int {

pub fn new(i: u32) -> Self {

Self(i)

}

pub fn to\_u32(&self) -> u32 {

self.0

}

pub fn to\_usize(&self) -> usize {

self.0 as \_

}

}

impl std::cmp::PartialEq<u32> for Int {

#[inline]

fn eq(&self, other: &u32) -> bool {

self.0 == \*other

}

}

impl std::cmp::PartialEq<usize> for Int {

#[inline]

fn eq(&self, other: &usize) -> bool {

self.0 as usize == \*other

}

}

#[derive(Clone, Debug, PartialEq, is\_macro::Is)]

pub enum Constant {

None,

Bool(bool),

Str(String),

Bytes(Vec<u8>),

Int(BigInt),

Tuple(Vec<Constant>),

Float(f64),

Complex { real: f64, imag: f64 },

Ellipsis,

}

impl Constant {

pub fn is\_true(self) -> bool {

self.bool().map\_or(false, |b| b)

}

pub fn is\_false(self) -> bool {

self.bool().map\_or(false, |b| !b)

}

pub fn complex(self) -> Option<(f64, f64)> {

match self {

Constant::Complex { real, imag } => Some((real, imag)),

\_ => None,

}

}

}

impl From<String> for Constant {

fn from(s: String) -> Constant {

Self::Str(s)

}

}

impl From<Vec<u8>> for Constant {

fn from(b: Vec<u8>) -> Constant {

Self::Bytes(b)

}

}

impl From<bool> for Constant {

fn from(b: bool) -> Constant {

Self::Bool(b)

}

}

impl From<BigInt> for Constant {

fn from(i: BigInt) -> Constant {

Self::Int(i)

}

}

#[cfg(feature = "rustpython-literal")]

impl std::fmt::Display for Constant {

fn fmt(&self, f: &mut std::fmt::Formatter<'\_>) -> std::fmt::Result {

match self {

Constant::None => f.pad("None"),

Constant::Bool(b) => f.pad(if \*b { "True" } else { "False" }),

Constant::Str(s) => rustpython\_literal::escape::UnicodeEscape::new\_repr(s.as\_str())

.str\_repr()

.write(f),

Constant::Bytes(b) => {

let escape = rustpython\_literal::escape::AsciiEscape::new\_repr(b);

let repr = escape.bytes\_repr().to\_string().unwrap();

f.pad(&repr)

}

Constant::Int(i) => i.fmt(f),

Constant::Tuple(tup) => {

if let [elt] = &\*\*tup {

write!(f, "({elt},)")

} else {

f.write\_str("(")?;

for (i, elt) in tup.iter().enumerate() {

if i != 0 {

f.write\_str(", ")?;

}

elt.fmt(f)?;

}

f.write\_str(")")

}

}

Constant::Float(fp) => f.pad(&rustpython\_literal::float::to\_string(\*fp)),

Constant::Complex { real, imag } => {

if \*real == 0.0 {

write!(f, "{imag}j")

} else {

write!(f, "({real}{imag:+}j)")

}

}

Constant::Ellipsis => f.pad("..."),

}

}

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn test\_is\_macro() {

let none = Constant::None;

assert!(none.is\_none());

assert!(!none.is\_bool());

}

}