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 №4

на тему

**СЕМАНТИЧЕСКИЙ 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 семантическими анализаторами, разработка семантического анализатора подмножества языка программирования, определенного в лабораторной работе 1, определение семантических правил.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Разработанный программный продукт представляет собой семантический анализатор для языка программирования Python. Он просматривает абстрактное синтаксическое дерево, построенное синтаксическим анализатором на наличие ошибок типизации.

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов(coercion)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным. Семантический анализ – это третья фаза работы компилятора. Семантический анализ гарантирует, что объявления и инструкции программы семантически корректны. Это набор процедур, который вызывается синтаксическим анализатором так, как и когда этого требует грамматика. Для проверки согласованности данного кода используются как синтаксическое дерево предыдущего этапа, так и таблица символов. Проверка типов является важной частью семантического анализа, когда компилятор проверяет, что каждый оператор имеет совпадающие операнды [1].

1. **РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**

В рамках лабораторной работы был разработан семантический анализатор языка Python, для тестирования был создан файл, который содержит основные конструкции языка и некоторые типовые семантические ошибки (рисунок 1).

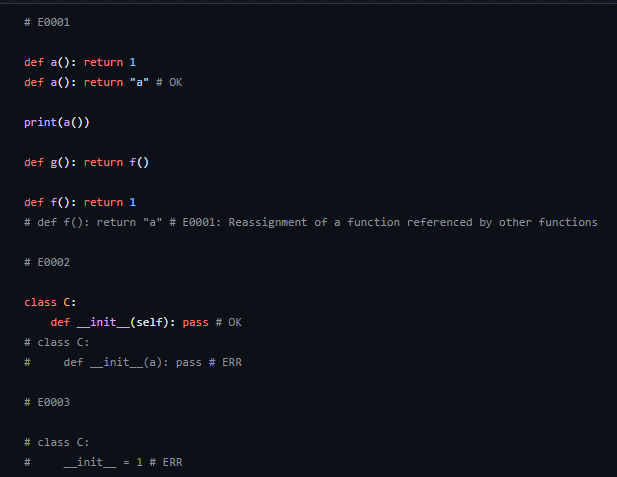


Рисунок 1 – Тестовый код

После того, как на этот код был запущен семантический анализатор, можно увидеть результаты выполнения программы. На следующем рисунке приведен вывод обработки кода семантическим анализатором, который указывает на неверное объявление стандартных функций класса (рисунок 2).

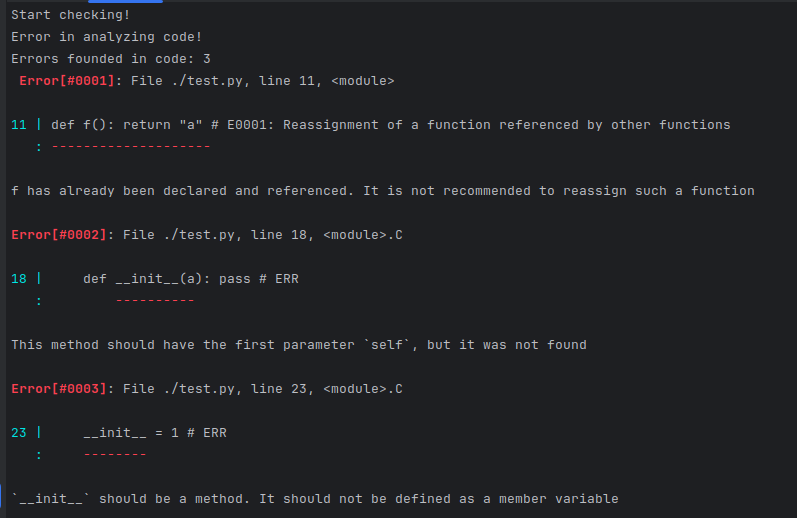


Рисунок 2 – Вывод ошибок

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы был разработан семантический анализатор подмножества языка программирования Python, который в результате своей работы выводит сообщение о правильности кода, либо же список семантических ошибок. Был продемонстрирован результат работы программы при обнаружении ошибок.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Семантический анализ при разработке компилятора [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/. – Дата доступа: 05.04.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – analyze.rs

use erg\_common::config::ErgConfig;

use erg\_common::error::{ErrorCore, ErrorKind, MultiErrorDisplay};

use erg\_common::style::colors::{BLUE, GREEN, RED, YELLOW};

use erg\_common::style::RESET;

use erg\_common::traits::{New, ExitStatus, Runnable, Stream};

use erg\_common::Str;

use erg\_compiler::GenericHIRBuilder;

use erg\_compiler::artifact::{BuildRunnable, Buildable, CompleteArtifact, IncompleteArtifact};

use erg\_compiler::build\_package::{GenericPackageBuilder, CheckStatus};

use erg\_compiler::erg\_parser::parse::Parsable;

use erg\_compiler::error::{CompileError, CompileErrors};

use erg\_compiler::module::SharedCompilerResource;

use py2erg::{dump\_decl\_er, reserve\_decl\_er, ShadowingMode};

use rustpython\_ast::source\_code::{RandomLocator, SourceRange};

use rustpython\_ast::{Fold, ModModule};

use rustpython\_parser::{Parse, ParseErrorType};

use crate::handle\_err;

#[derive(Debug, Default)]

pub struct SimplePythonParser {

cfg: ErgConfig,

}

impl Parsable for SimplePythonParser {

fn parse(code: String) -> Result<ParseArtifact, IncompleteParseArtifact<Module, ParseErrors>> {

let mut slf = Self::new(ErgConfig::string(code.clone()));

slf.build\_ast(code)

.map(|art| {

ParseArtifact::new(art.ast.module, art.warns.into())

})

.map\_err(|iart| {

IncompleteParseArtifact::new(

iart.ast.map(|art| art.module),

iart.errors.into(),

iart.warns.into(),

)

})

}

}

impl New for SimplePythonParser {

fn new(cfg: ErgConfig) -> Self {

Self { cfg }

}

}

impl ASTBuildable for SimplePythonParser {

fn build\_ast(

&mut self,

code: String,

) -> Result<ParseArtifact<AST, ParserRunnerErrors>, IncompleteParseArtifact<AST, ParserRunnerErrors>> {

let filename = self.cfg.input.filename();

let py\_program = self.parse\_py\_code(code)?;

let shadowing = if cfg!(feature = "debug") {

ShadowingMode::Visible

} else {

ShadowingMode::Invisible

};

let converter = py2erg::ASTConverter::new(ErgConfig::default(), shadowing);

let IncompleteArtifact{ object: Some(erg\_module), errors, warns } = converter.convert\_program(py\_program) else { unreachable!() };

let erg\_ast = AST::new(erg\_common::Str::rc(&filename), erg\_module);

if errors.is\_empty() {

Ok(ParseArtifact::new(erg\_ast, warns.into()))

} else {

Err(IncompleteParseArtifact::new(

Some(erg\_ast),

errors.into(),

warns.into(),

))

}

}

}

impl SimplePythonParser {

pub fn parse\_py\_code(&self, code: String) -> Result<ModModule<SourceRange>, IncompleteParseArtifact<AST, ParserRunnerErrors>>{

let py\_program = ModModule::parse(&code, "<stdin>").map\_err(|err| {

let mut locator = RandomLocator::new(&code);

// let mut locator = LinearLocator::new(&py\_code);

let err = locator.locate\_error::<\_, ParseErrorType>(err);

let msg = err.to\_string();

let loc = err.location.unwrap();

let core = ErrorCore::new(

vec![],

msg,

0,

ErrorKind::SyntaxError,

erg\_common::error::Location::range(

loc.row.get(),

loc.column.to\_zero\_indexed(),

loc.row.get(),

loc.column.to\_zero\_indexed(),

),

);

let err = CompileError::new(core, self.cfg.input.clone(), "".into());

IncompleteParseArtifact::new(None, ParserRunnerErrors::from(err), ParserRunnerErrors::empty())

})?;

let mut locator = RandomLocator::new(&code);

// let mut locator = LinearLocator::new(&code);

let module = locator

.fold(py\_program)

.map\_err(|\_err| ParserRunnerErrors::empty())?;

Ok(module)

}

}

#[derive(Debug, Default)]

pub struct PythonAnalyzer {

pub cfg: ErgConfig,

checker: GenericPackageBuilder<SimplePythonParser, GenericHIRBuilder<SimplePythonParser>>,

}

impl New for PythonAnalyzer {

fn new(cfg: ErgConfig) -> Self {

let checker = GenericPackageBuilder::new(cfg.clone(), SharedCompilerResource::new(cfg.clone()));

Self { checker, cfg }

}

}

impl Runnable for PythonAnalyzer {

type Err = CompileError;

type Errs = CompileErrors;

const NAME: &'static str = "Python Analyzer";

#[inline]

fn cfg(&self) -> &ErgConfig {

&self.cfg

}

#[inline]

fn cfg\_mut(&mut self) -> &mut ErgConfig {

&mut self.cfg

}

fn finish(&mut self) {

self.checker.finish();

}

fn initialize(&mut self) {

self.checker.initialize();

}

fn clear(&mut self) {

self.checker.clear();

}

fn eval(&mut self, src: String) -> Result<String, Self::Errs> {

self.checker.eval(src)

}

fn exec(&mut self) -> Result<ExitStatus, Self::Errs> {

self.checker.exec()

}

}

impl Buildable for PythonAnalyzer {

fn inherit(cfg: ErgConfig, shared: SharedCompilerResource) -> Self {

let mod\_name = Str::rc(&cfg.input.file\_stem());

Self {

cfg: cfg.copy(),

checker: GenericPackageBuilder::new\_with\_cache(cfg, mod\_name, shared),

}

}

fn inherit\_with\_name(cfg: ErgConfig, mod\_name: Str, shared: SharedCompilerResource) -> Self {

Self {

cfg: cfg.copy(),

checker: GenericPackageBuilder::new\_with\_cache(cfg, mod\_name, shared),

}

}

fn build(&mut self, code: String, mode: &str) -> Result<CompleteArtifact, IncompleteArtifact> {

self.analyze(code, mode)

}

fn build\_from\_ast(

&mut self,

ast: AST,

mode: &str,

) -> Result<CompleteArtifact<erg\_compiler::hir::HIR>, IncompleteArtifact<erg\_compiler::hir::HIR>> {

self.check(ast, CompileErrors::empty(), CompileErrors::empty(), mode)

}

fn pop\_context(&mut self) -> Option<ModuleContext> {

self.checker.pop\_context()

}

fn get\_context(&self) -> Option<&ModuleContext> {

self.checker.get\_context()

}

}

impl BuildRunnable for PythonAnalyzer {}

impl PythonAnalyzer {

pub fn new(cfg: ErgConfig) -> Self {

New::new(cfg)

}

fn check(&mut self, erg\_ast: AST, mut errors: CompileErrors, mut warns: CompileErrors, mode: &str) -> Result<CompleteArtifact, IncompleteArtifact> {

match self.checker.build\_from\_ast(erg\_ast, mode) {

Ok(mut artifact) => {

artifact.warns.extend(warns);

artifact.warns =

handle\_err::filter\_errors(self.get\_context().unwrap(), artifact.warns);

if errors.is\_empty() {

Ok(artifact)

} else {

Err(IncompleteArtifact::new(

Some(artifact.object),

errors,

artifact.warns,

))

}

}

Err(iart) => {

Err(IncompleteArtifact::new(iart.object, errors, warns))

}

}

}

pub fn analyze(

&mut self,

py\_code: String,

mode: &str,

) -> Result<CompleteArtifact, IncompleteArtifact> {

let filename = self.cfg.input.filename();

let parser = SimplePythonParser::new(self.cfg.copy());

let py\_program = parser.parse\_py\_code(py\_code)

.map\_err(|iart| {

IncompleteArtifact::new(

None,

iart.errors.into(),

iart.warns.into(),

)

})?;

let shadowing = if cfg!(feature = "debug") {

ShadowingMode::Visible

} else {

ShadowingMode::Invisible

};

let converter = py2erg::ASTConverter::new(self.cfg.copy(), shadowing);

let IncompleteArtifact{ object: Some(erg\_module), errors, warns } = converter.convert\_program(py\_program) else { unreachable!() };

let erg\_ast = AST::new(erg\_common::Str::rc(&filename), erg\_module);

erg\_common::log!("AST:\n{erg\_ast}");

self.check(erg\_ast, errors, warns, mode)

}

pub fn run(&mut self) {

if self.cfg.dist\_dir.is\_some() {

reserve\_decl\_er(self.cfg.input.clone());

}

let py\_code = self.cfg.input.read();

let filename = self.cfg.input.filename();

println!("{BLUE}Start checking{RESET}: {filename}");

match self.analyze(py\_code, "exec") {

Ok(artifact) => {

if !artifact.warns.is\_empty() {

println!(

"{YELLOW}Found {} warnings{RESET}: {}",

artifact.warns.len(),

self.cfg.input.filename()

);

artifact.warns.write\_all\_stderr();

}

println!("{GREEN}All checks OK{RESET}: {}", self.cfg.input.filename());

if self.cfg.dist\_dir.is\_some() {

dump\_decl\_er(

self.cfg.input.clone(),

artifact.object,

CheckStatus::Succeed,

);

println!("A declaration file has been generated to \_\_pycache\_\_ directory.");

}

std::process::exit(0);

}

Err(artifact) => {

if !artifact.warns.is\_empty() {

println!(

"{YELLOW}Found {} warnings{RESET}: {}",

artifact.warns.len(),

self.cfg.input.filename()

);

artifact.warns.write\_all\_stderr();

}

let code = if artifact.errors.is\_empty() {

println!("{GREEN}All checks OK{RESET}: {}", self.cfg.input.filename());

0

} else {

println!(

"{RED}Found {} errors{RESET}: {}",

artifact.errors.len(),

self.cfg.input.filename()

);

artifact.errors.write\_all\_stderr();

1

};

if self.cfg.dist\_dir.is\_some() {

dump\_decl\_er(

self.cfg.input.clone(),

artifact.object.unwrap(),

CheckStatus::Failed,

);

println!("A declaration file has been generated to \_\_pycache\_\_ directory.");

}

std::process::exit(code);

}

}

}

}