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 анализатор»

Выполнил             К. А. Тимофеев

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 9](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 10](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка собственного семантического анализатора для языка программирования С. Необходимо вывести результат анализа и обработать возможные семантические ошибки.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

К этапам трансляции относятся следующие этапы:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– оптимизация;

– генерация кода.

На этапе генерации компилятор создает код, который представляет собой набор инструкций, понятных для целевой аппаратной платформы, итоговый файл компилируется в исполняемый файл, который может быть запущен на целевой платформе без необходимости наличия кода.

Фаза эмуляции интерпретатора происходит во время выполнения программы. В отличие от компилятора, интерпретатор работает с кодом напрямую, без предварительной генерации машинного кода.

Лексический анализатор – первый этап трансляции. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в лексемы или значащие последовательности. Лексема – это элементарная единица, которая может являться ключевым словом, идентификатором, константным значением. Для каждой лексемы анализатор строит токен, который по сути является кортежем, содержащим имя и значение.[1]

Синтаксический анализатор выясняет, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики языка программирования. Синтаксический анализатор получает на вход результат лексического анализатора и разбирает его в соответствии с грамматикой. Результат синтаксического анализа обычно представляется в виде синтаксического дерева разбора.[2]

Семантический анализ обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка.

В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных, то есть для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение.

Число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры.

Абстрактное синтаксическое дерево конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в синтаксических анализаторах для промежуточного представления программы между деревом разбора (деревом с конкретным синтаксисом) и структурой данных, которая затем используется в качестве внутреннего представления в компиляторе или интерпретаторе программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в абстрактном синтаксическом дереве группировка операндов явно задаётся структурой дерева.

Для языка, который описывается контекстно-свободной грамматикой создание дерева в синтаксическом анализаторе является тривиальной задачей. Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в дерево, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, которые не используются в абстрактном синтаксисе, для получения абстрактного синтаксического дерева.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы был реализован конечный вид анализатора кода, который включает в себя лексический, синтаксический и семантический анализы. Были совершены проверки на такие типы ошибок как:

– объявление одноименных переменных или функций в одной области видимости;

– несовпадение параметров и аргументов при вызове функции;

– неверное преобразование типов данных;

– неверное применение закрывающихся одинарных и двойных кавычек;

При неверном преобразовании типов данных, когда указателю присваивается, например, значение с плавающей точкой, будет выведена ошибка об этом. Пример семантической ошибки при неверном преобразовании типов данных представлен на рисунке 3.1.



Рисунок 3.1 – Ошибка при неверном преобразовании типов данных

При несовпадении количества параметров и аргументов при вызове функции с учетом того, что параметрам функции не присваивается значение, также будет выведена семантическая ошибка. Пример тестового кода с ошибкой данного типа представлен на рисунке 3.2.

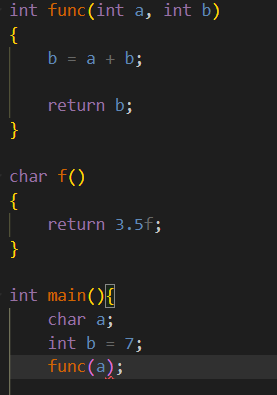


Рисунок 3.2 – Пример тестового кода

Пример семантической ошибки при несовпадении количества параметров и аргументов представлен на рисунке 3.3.



Рисунок 3.3 – Ошибка при различном количестве параметров и аргументов

Таким образом в ходе данной лабораторной работы был организован полноценный анализатор кода, который включает в себя лексический, синтаксический и семантические анализы.

# **ВЫВОДЫ**

В ходе лабораторной работы был реализован семантический анализатор, основанный на результатах синтаксического анализатора. В итоге был получен полный анализатор кода программ на языке С, включающий в себя лексический, синтаксический и семантический анализы.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.
2. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код parser.py

def ValidateTypes(self) -> MyType:

if self.initialValue:

t = self.initialValue.ValidateTypes()

if self.varType in types:

if t not in types:

raise Exception(f"Semantic error: declaration init value incorrect type: "+

f"init value type: {t.name}, variable type {self.varType.name}")

else:

if t != self.varType:

raise Exception(

f"Semantic error: declaration init value incorrect type: "

+ f"init value type: {t.name}, variable type {self.varType.name}"

)

def ValidateTypes(self) -> MyType:

for instruction in self.instructions:

instruction.ValidateTypes()

def ValidateTypes(self) -> MyType:

self.body.ValidateTypes()

def ValidateTypes(self) -> MyType:

program:Program = self.getRoot()

func = program.functions.get(self.func)

if func is None:

raise Exception(f"Semantic error: there is no function {self.func.name}")

args = func.argsDeclarations

if len(args) != len(self.args):

raise Exception(f'Semantic error: incorrect number of arguments in call of {func.name}'+

f'({len(self.args)} instead of {len(args)})')

for i in range(len(args)):

t = self.args[i].ValidateTypes()

if args[i].type not in types:

if t != args[i].type:

raise Exception(f'Semantic error: incorrect type of argument {i} in function call of {func.name}')

else:

if t not in types:

raise Exception(

f"Semantic error: incorrect type of argument {i} in function call of {func.name}"

)

return func.mytype

def ValidateTypes(self) -> MyType:

for f in self.functions.values():

f.ValidateTypes()

for v in self.variables.values():

v.ValidateTypes()

def ValidateTypes(self) -> MyType:

if self.operator != operatorsDict['\*'] and self.operator != operatorsDict['&']:

self.operator.verificator(self.operator, self.variable.mytype, None)

else:

if self.operator is operatorsDict['\*']:

if type(self.variable.mytype) != MyPtr:

raise Exception(

f"Semantic error: attempt to use {self.variable.mytype.name} as pointer"

)

ptr: MyPtr = self.variable.mytype

if ptr.level == 1:

return ptr.pointersTo

else:

return MyPtr([ptr.pointersTo], ptr.name[i:-1])

else:

if type(self.variable.mytype) != MyPtr:

return MyPtr([self.variable.mytype], self.variable.mytype.name + '\*')

else:

return MyPtr([self.variable.mytype.pointersTo], self.variable.mytype.name + '\*')

return self.variable.mytype

def ValidateTypes(self) -> MyType:

tl = self.left.ValidateTypes()

tr = self.right.ValidateTypes()

print(self.operator.symbol)

self.operator.verificator(self.operator, tl, tr)

if self.operator.prior != 15:

if tl.prior < tr.prior:

return tl

else:

return tr

else:

return tr

def ValidateTypes(self) -> MyType:

it = self.parent

while type(it) != FunctionNode:

it = it.parent

retType = self.retVal.ValidateTypes()

it:FunctionNode = it

if it.mytype not in types:

if it.mytype != retType:

raise Exception(f"Semantic Error: incorrect return type in function {it.name}")

return retType