Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

“Семантический анализатор”

Выполнил:

студент гр. 853504

Кузьма В. В.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2021

**Содержание**

[**1. Цель работы 3**](#_Toc64740995)

[**2. Краткие теоритические сведения 5**](#_Toc64740996)

[**3. Семантические ошибки 8**](#_Toc64741001)

[**Приложение. Текст программ 9**](#_Toc64741002)

# **Цель работы**

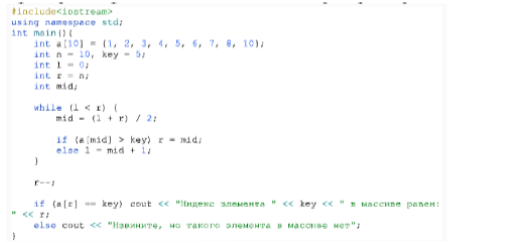
В процессе **семантического анализа** проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – **генерации кода**. При семантическом анализе используются **иерархические структуры**, полученные во время синтаксического анализа для идентификации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

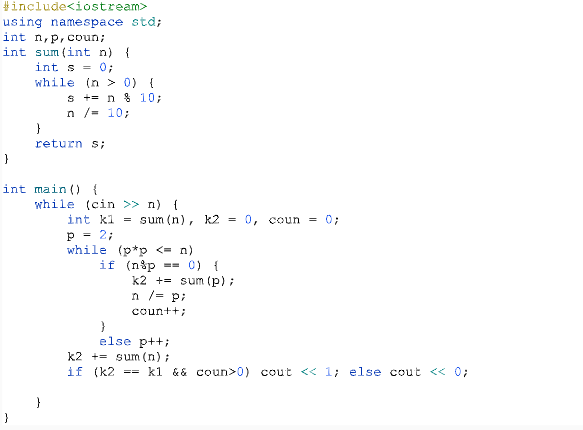
Важным аспектом семантического анализа является **проверка типов**, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива 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.

В языках со статическими типами, например **С**, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (*в процессе компиляции, а не во время выполнения программы!*) многие программные ошибки.

В качестве тестового примера возьмём два примера из **1-ой лабораторной работы**:





# **Краткие теоритические сведения**

**Семантический анализ** является центральной фазой трансляции, связывающей 2 ее логические части: **анализ исходной программы** и **синтез объектной программы**. На этапе семантического анализа обрабатываются **программные конструкции**, распознанные синтаксическим анализатором.

**Идентификация идентификаторов** – одна из задач, решение которой необходимо для проверки правильности использования типов.

Понятно, что невозможно убедиться в правильности использования типов в какой-нибудь конструкции до тех пор, пока не определим типы всех ее составных частей. Например, для того, чтобы выяснить правильность оператора присваивания, мы должны знать типы его получателя (левой части) и источника (правой части). Для того, чтобы выяснить, каков тип идентификатора, являющегося, например, получателем присваивания, надо понять, каким образом этот идентификатор был объявлен в программе.

Каждое вхождение идентификатора в программу является либо определяющим, либо использующим. Под **определяющим вхождением** идентификатора понимается его вхождение в описание, например, int i.

Все остальные **вхождения** являются **использующими** (i = 5; i + 13).

**Цель идентификации идентификаторов** – определить тип использующего вхождения идентификатора. Эта задача может быть полностью или частично решена в *фазе синтаксического анализа*. Все зависит от того, может ли использующее вхождение идентификатора встретиться в программе до определяющего вхождения, или нет.

Если все *определяющие вхождения идентификаторов* должны быть расположены текстуально перед использующими вхождениями, то можно выполнить идентификацию на фазе синтаксического анализа. Если же нет, то в фазе синтаксического анализа мы можем обработать определяющие вхождения идентификаторов и только на следующем просмотре текста программы выполнить собственно **идентификацию**.

Вне зависимости от того, на каком просмотре будет выполняться идентификация идентификаторов, при обработке определяющего вхождения идентификатора необходимо запомнить информацию о типе этого идентификатора.

*Это можно сделать несколькими путями:*

• **создать узел** в синтаксическом дереве для конструкции *«описание идентификатора»* и запоминать информацию о типе идентификатора в этом узле;

• **создать таблицу** идентификаторов (*IdTab*) и в ней запоминать информацию о тип идентификатора;

Если **контроль типов** осуществляется во время трансляции программы, то мы говорим о статическом контроле типов, в противном случае, т. е. если контроль типов производится во время исполнения объектной программы, мы говорим о динамическом контроле типов. В принципе, контроль типов всегда может выполняться **динамически**, если в объектном коде вместе со значением будет размещаться и тип этого значения.

Понятно, что **динамический контроль** типов приводит к увеличению размера и времени исполнения объектной программы, и уменьшению ее надежности. Язык программирования называется языком со **статическим контролем типов** или **строго типизированным языком** (*strongly typed language*), если тип любого выражения может быть определен во время трансляции, т. е. если можно гарантировать, что объектная программа выполняется без типовых ошибок. К числу строго типизированных языков относится, например, **Pascal**.

Необходимой частью контроля типов является **проверка эквивалентности** типов (*equivalence of types*). Крайне необходимо, чтобы компилятор выполнял проверку эквивалентности типов быстро.

**Структурная эквивалентность типов** (*Structural equivalence of types*). Два типа называются эквивалентными если они являются одинаковыми примитивными типами, либо они были сконструированы с применением одного и того же конструктора к структурно эквивалентным типам. Иными словами, два типа структурно эквивалентны тогда и только тогда, когда они идентичны.

Второй подход связан с понятием **эквивалентности имен** (*name equivalence*). В этом случае каждое имя типа рассматривается как уникальный тип, таким образом, два имени типов эквивалентны, если они идентичны.

**Основные функции семантического анализатора:**

1) ***Заполнение таблиц имен***. Таблица формируется на этапе лексического анализа, где в нее помещаются все уникальные имена, распознанные сканером. Во время семантического анализа для каждого имени заносятся все данные, полученные из текста программы (тип идентификатора, тип значений и т.д.).

2) ***Выделение неявно заданной информации***. В представлении программ некоторые данные об элементах программы не указаны явно.

3*)****Обнаружение ошибок****.* Синтаксический анализ определяет корректность отдельных конструкций и программы в целом с точки зрения формальных правил используемого языка, но и здесь могут быть ошибки (не согласованы типы правой и левой частей оператора присваивания, несколько одинаковых меток и т.д.).

4) ***Выполнение некоторых операций программы*.** Присваивание начальных значений; Действия с константами; Обработка директив компилятора.

5) ***Формирование внутренней формы программы***. Часто используются такие формы, как семантическое дерево, польская запись, список тетрад.

## **Семантические ошибки**

Добавим специально *семантические ошибки* в нашу программу для просмотра обработки ошибочных ситуаций в код из 1-ой лабораторной работы:

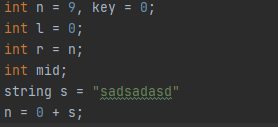
1. Ошибка – присвоение неверному типу





Рис 1. Неверное присвоение типа

1. Ошибка – операция между двумя несовместными типами



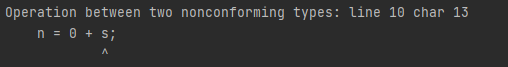


Рис 2. Существующая переменная

1. Ошибка – недостаточное количество аргументов в функции.





Рис 3. Недостаточное количество аргументов функции.

**Приложение. Текст программ**

import re  
from ply.lex import LexToken  
  
from Tokenizer import tokens, lexer  
from ply import yacc  
  
  
  
  
class Node:  
 def parts\_str(self):  
 st = []  
 for part in self.parts:  
 st.append(str(part))  
 return "\n".join(st)  
  
 def \_\_repr\_\_(self):  
 return self.type + ":\n\t" + self.parts\_str().replace("\n", "\n\t")  
  
 def add\_parts(self, parts):  
 self.parts += parts  
 return self  
  
 def \_\_init\_\_(self, type, parts):  
 self.type = type  
 self.parts = parts  
  
start = 'function'  
  
variables = []  
  
  
  
def token\_in\_stack(name, stack):  
 for item in stack:  
 if isinstance(item, LexToken) and item.value == name:  
 return True  
 return False  
  
  
def p\_cout(p):  
 *'''cout : COUT LESSS args semicolons'''* p[0] = Node('cout', p[3:])  
  
def p\_initlist(p):  
 *'''init\_list : LFIGURPAREN args RFIGURPAREN'''* p[0] = Node('init\_list', p[1:])  
  
def p\_function(p):  
 *'''function : func\_header func\_body  
  
 '''* p[0] = Node('function', p[1:])  
  
  
def p\_func\_header(p):  
 *'''func\_header :  
 | FUNCTION VARIABLE LPAREN args RPAREN  
 | INTEGER VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_declaration', [p[2], p[4]])  
  
  
def p\_args(p):  
 *'''args :  
 | expr  
 | args COMMA expr  
 | args LESSS expr  
 | args LESS expr semicolons  
 | LESSS args LESS expr'''* if len(p) <= 2:  
 p[0] = Node('args', p[1:] if p[1:] else ['EMPTY'])  
 else:  
 p[0] = p[1].add\_parts([p[3]])  
  
  
def p\_func\_body(p):  
 *'''func\_body : block'''* p[0] = p[1]  
  
def p\_if\_s(p):  
 *'''if\_s : IF LPAREN condition RPAREN block  
 | IF LPAREN condition RPAREN line semicolons  
 | IF LPAREN condition RPAREN cout'''* p[0] = Node('if', [p[3], p[5]])  
  
def p\_block(p):  
 *'''block : LFIGURPAREN body RFIGURPAREN'''* p[0] = Node('block', [p[2]])  
  
  
def p\_body(p):  
 *'''body :  
 | body line semicolons  
 | body multiline'''* if len(p) > 1:  
 if p[1] is None:  
 p[1] = Node('body', [])  
 p[0] = p[1].add\_parts([p[2]])  
 else:  
 p[0] = Node('body', [])  
  
  
def p\_semicolons(p):  
 *'''semicolons : SEMICOLON  
 | semicolons SEMICOLON'''*def p\_multiline(p):  
 *'''multiline : while\_statement  
 | if\_s  
 | else'''* p[0] = p[1]  
  
  
def p\_line(p):  
 *'''line : assign  
 | func  
 | RETURN expr  
 | RETURN func'''* if len(p) == 3:  
 p[0] = Node('return', [p[2]])  
 else:  
 p[0] = p[1]  
  
  
def p\_else(p):  
 *'''else : ELSE block  
 | ELSE line semicolons  
 | ELSE cout'''* p[0] = Node('else', [p[2]])  
  
def p\_while\_statement(p):  
 *'''while\_statement : WHILE LPAREN condition RPAREN block  
 | WHILE LPAREN condition RPAREN line semicolons'''* p[0] = Node('while', [p[3], p[5]])  
  
  
def p\_condition(p):  
 *'''condition : expr cond\_sign expr'''* p[0] = Node('condition', [p[1], p[2], p[3]])  
  
  
def p\_cond\_sign(p):  
 *'''cond\_sign : NOTEQUAL  
 | EQ  
 | LESS  
 | MORE  
 '''* p[0] = p[1]  
  
def p\_assign\_array(p):  
 *'''assign : INTEGER variable LSQRPAREN INT RSQRPAREN OPERATOR EQUAL expr  
 '''* if len(p) == 5:  
 value = p[2]  
 if value in variables:  
 raise VariableOverridingError(value, lexer.lineno)  
 else:  
 variables.append(value)  
 p[0] = Node('assign\_array', [p[2], p[4]])  
 else:  
 value = p[1]  
 if value not in variables:  
 raise NotAssignedVariableError(value, lexer.lineno)  
 p[0] = Node('assign\_array', [p[1], p[3]])  
  
def p\_assign(p):  
 *'''assign : variable EQUAL expr  
 | VAR variable EQUAL expr  
 | VAR variable EQUAL function  
 | INTEGER variable EQUAL expr  
 | INTEGER variable EQUAL function  
 | INTEGER variable EQUAL init\_list'''* if len(p) == 5:  
 value = p[2]  
 if value in variables:  
 raise VariableOverridingError(value, lexer.lineno)  
 else:  
 variables.append(value)  
 p[0] = Node('assign', [p[2], p[4]])  
 else:  
 value = p[1]  
 if value not in variables and '[' not in value and '"' not in value:  
 raise NotAssignedVariableError(value, lexer.lineno)  
 p[0] = Node('assign', [p[1], p[3]])  
  
  
  
  
def p\_func(p):  
 *'''func : VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_call', [p[1], p[3]])  
  
  
def p\_expr(p):  
 *'''expr : fact  
 | expr OPERATOR fact'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_fact(p):  
 *'''fact : term  
 | fact OPERATOR term'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
  
  
def p\_term(p):  
 *'''term : arg  
 | LPAREN expr RPAREN'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = p[2]  
  
  
def p\_arg(p):  
 *'''arg : STRING  
 | variable  
 | INT  
 | FLOAT  
 | method  
 | func  
 | STRINGLIT'''* if len(p) == 3:  
 p[1], p[2] = p[2], p[1]  
 value = p[1]  
 if isinstance(value, str):  
 if token\_in\_stack('function', p.stack):  
 variables.append(value)  
 elif value not in variables and '[' not in value and '"' not in value:  
  
 raise NotAssignedVariableError(value, lexer.lineno)  
 p[0] = Node('arg', [p[1]])  
  
  
def p\_method(p):  
 *'''method : variable\_or\_arg DOT func'''* p[0] = Node('method', [p[1], p[3]])  
  
  
def p\_variable\_or\_arg(p):  
 *'''variable\_or\_arg : variable  
 | arg'''* p[0] = p[1]  
  
  
def p\_variable(p):  
 *"""variable : VARIABLE"""* p[0] = p[1]  
  
parser\_errors = []  
  
def p\_error(p):  
 if p.value == 'include':  
 return  
 print('Unexpected token in line %d: %s' % (p.lineno, p))  
  
  
  
  
parser = yacc.yacc()  
  
  
def build\_tree(code):  
 p = parser.parse(code)  
 return p  
  
operations = {  
 '+': lambda x, y: x + y,  
 '-': lambda x, y: x - y,  
 '\*': lambda x, y: x \* y,  
 '/': lambda x, y: x / y,  
}  
  
\_\_variables = {}  
  
def parse\_tree(tree):  
 try:  
 tree\_type = tree.type  
 parts = tree.parts  
 except AttributeError:  
 return tree  
 if tree\_type == 'assign':  
 \_\_variables[parts[0]] = parse\_tree(parts[1])  
 return  
 if tree\_type == 'arg':  
 arg = parts[0]  
 if isinstance(arg, int):  
 return arg  
 elif arg in \_\_variables:  
 return \_\_variables[arg]  
 else:  
 return 'method'  
 if tree\_type in operations:  
 first = parse\_tree(parts[0])  
 second = parse\_tree(parts[1])  
 if tree\_type == '/' and second == 0:  
 raise CustomZeroDivisionError(11)  
 return operations[tree\_type](first, second)  
 for part in parts:  
 parse\_tree(part)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with open("main.cpp") as f:  
 lines = f.readlines()  
 with open("main.cpp") as f:  
 characters = f.read()  
  
 try:  
 tree = build\_tree(characters)  
 parse\_tree(tree)  
 except CustomException as e:  
 print(e)  
 k = str(e).split()  
 if k[0] == "WrongTypeError":  
 print(f"Assigning the wrong type: line {k[3]} char {lines[int(k[3])].find(k[6][1:-1])}")  
 print(lines[int(k[3])][:-1])  
 print(' ' \* lines[int(k[3])].find('/') + '^')  
 if k[0] == "TypeOperationError":  
 print(f"operation between two nonconforming types: line {k[3]} char {lines[int(k[3]) - 1].find(k[6][1:-1])}")  
 print(lines[int(k[3]) - 1][:-1])  
 print(' ' \* lines[int(k[3]) - 1].find(k[6][1:-1]) + '^')  
 if k[0] == "VariableOverridingError":  
 print(f"Variable declared: line {k[3]} char {lines[int(k[3]) - 1].find(k[6][1:-1])}")  
 print(lines[int(k[3]) - 1][:-1])  
 print(' ' \* lines[int(k[3]) - 1].find(k[6][1:-1]) + '^')

if k[0] == "ArgumentLessError":  
 print(f"Insuffient number of agruments({k[3]}) arguments( {lines[int(k[3]) - 1].find(k[6][1:-1])} line {k[4]} char {k[5]}")  
 print(lines[int(k[4]))  
 print(' ' \* lines[int(k[4]) - 1].find(k[6][1:-1]) + '^')  
 if k[0] == "ArgumentMoreError":  
 print(  
 f"Excessive number of agruments({k[3]}) arguments( {lines[int(k[3]) - 1].find(k[6][1:-1])} line {k[4]} char {k[5]}")  
 print(lines[int(k[4]))  
 print(' ' \* lines[int(k[4]) - 1].find(k[6][1:-1]) + '^')

for error in parser\_errors:  
 print(error)