Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

По теме “Семантический анализатор”

Выполнил:
студент гр. 853504
Шевченя И.В.
Проверил:
Ст. преподаватель КИ Шиманский В. В.

Минск 2021

1 Постановка задачи:

Разработка семантического анализатора подмножества языка программирования, определенного в лабораторной работе 1, для реализации возможности интерпретации программы на выбранном языке. В данной работе ставится задача по исследованию области семантических анализаторов, рассмотрению аналогов и написанию своего собственного анализатора семантических ошибок, заложение в программу нескольких ошибок. Правильная их обработка означает готовность работы.

2 Теория:

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

Рисунок 1. Общая схема работы транслятора

Фаза контроля типов проверяет, удовлетворяет ли программа контекстным условиям. Главной составляющей контекстных условия является «правильное использование» программой типов данных, предоставляемых входным языком, т.е. корректность выражений,

встречающихся в программе, с точки зрения использования типов. Данная задача включает, в частности, нахождение объявления в программе каждого используемого идентификатора, и проверку корректности его появления в использующем контексте.

Идентификация идентификаторов – одна из задач, решение которой необходимо для проверки правильности использования типов. Понятно, что мы не можем 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, мы должны 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, например, *i = 5* или *i+13*.

Цель идентификации идентификаторов – определить тип использующего вхождения идентификатора. Эта задача может быть полностью или частично решена на 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 мы можем выполнить идентификацию на фазе синтаксического анализа. Если же нет, то на 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.

Основные функции семантического анализатора:

- 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.).

- Выделение неявно заданной информации. В представлении программ некоторые данные об элементах программы не указаны явно.
- 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.).
- Выполнение некоторых операций программы. Присваивание начальных значений; Действия с константами; Обработка директив компилятора.
- Формирование внутренней формы программы. Часто используются такие формы, как семантическое дерево, польская запись.

3. Результат работы анализатора:

Дерево программы, используемое для его обхода и поиска семантических ошибок.

Таблица переменных для каждой функции, полученная в результате выполнения программы:

```
{
'reverseArray': {
     'return': 'void',
     'args': {'arr': {'type': 'int', 'is_array': True}, 'start':
{'type': 'int', 'is_array': False}, 'end': {'type': 'int',
'is_array': False}},
     'additional_args': {'temp': {'type': 'int', 'value':
[<syntax_analysis.Node object at 0x7f3c19ccf040>]}}
},

'printArray': {
     'return': 'void',
     'args': {'arr': {'type': 'int', 'is_array': True}, 'size':
{'type': 'int', 'is_array': False}},
     'additional_args': {'i': {'type': 'int', 'value':
[<syntax_analysis.Node object at 0x7f3c19ccf730>]}}
},

'main': {
     'return': 'int',
     'args': {},
     'additional_args': {'arr': {'type': 'int', 'value': ['=',
<syntax_analysis.Node object at 0x7f3c19cd4190>]}, 'n': {'type':
'int', 'value': [<syntax_analysis.Node object at
0x7f3c19cd4790>]}, 'a': {'type': 'int', 'value':
[<syntax_analysis.Node object at 0x7f3c19cd4a00>]}, 'b':
{'type': 'int', 'value': [<syntax_analysis.Node object at
0x7f3c19cd4e20>]}, 't': {'type': 'int', 'value':
[<syntax_analysis.Node object at 0x7f3c19cd6130>]}}}
}
```

Рассмотрим текст программы с ошибками. При обнаружении их происходит вывод уведомления об ошибке (красным выделено место ошибки):

1. Отсутствие входной точки программы на языке C++ - функции main():

```
int main1()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    printArray(arr, n);

    reverseArray(arr, 0, n – 1);
…
}
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 331, in <module>
    raise Exception(
Exception: Program should have starting point as function with name 'main'
```

2. Вызов функции с неправильным числом параметров:

```
void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   {
       cout << arr[i];
       cout << " ";
   }
   cout << endl;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr);
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 196, in parse_tree
    raise Exception(
Exception: Wrong count of arguments passing to function printArray
```

3. Несравнимые типы операндов в условии:
   Сравнение строки и числа, т. е. сравнение «int» и «str»

```
void reverseArray(int arr[], int start, int end)
{
    while (start < "ILYA")
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 153, in parse_tree
    raise TypeError("Wrong operand types in condition")
TypeError: Wrong operand types in condition
```

4. Сложение операндов различных типов:

```
int a;
a = 10.2 + "10";
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 285, in parse_tree
    raise TypeError(
TypeError: Types mismatch: <class 'float'> and <class 'str'>
```

5. Инициализация переменной значением не соответствующим указанному значению:

```
int main()
{
    int ilya = "Shevchenya";
    int arr[] = {1, 2, 3, 4, 5, 6};
  …
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 324, in check_inits
    raise ValueError(
ValueError: Wrong initialization of variable: type str can't initialize variable 'ilya'
```

6. Использование при индексировании типов отличных от «int»:

```
void reverseArray(int arr[], int start, int end)
{
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[10.22];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

Код ошибки:

```
  File "/home/shevchenya/PycharmProjects/mtran/semantic_analysis.py", line 165, in parse_tree
    raise ValueError(f"Forbidden argument type in call {arg}")
ValueError: Forbidden argument type in call 10.22
```

Вывод

В ходе выполнения лабораторной была создана программа, позволяющая отслеживать ряд семантических ошибок в коде программ на языке C++. Программа проходит по синтаксическому дереву и осуществляет проверки. Если это не получается, то программа выдаёт соответствующую ошибку.

Приложение 1. Текст программы

Код семантического анализатора:

```
import re
from operator import add, mul, sub, truediv

from lexical_analysis import types
from syntax_analysis import build_tree


operations = "+-*/"
operation_resolver = {"+": add, "-": sub, "*": mul, "/":
truediv}
condition_resolver = ["<", ">", "<=", ">=", "==", "!="]
variables = {}
functions = {}


def parse_function_args(node):
    function_arguments = {}

    for part in node.parts:
        is_array = False

        if isinstance(part, str):
            break

        if len(part.parts) > 2:
            is_array = True

        function_arguments[part.parts[1]] = {
            "type": part.parts[0],
            "is_array": is_array,
        }

    return function_arguments


def parse_tree(tree):
    try:
        tree_type = tree.type
        parts = tree.parts
    except AttributeError:
        return tree

    if tree_type == "func_declaration":
        functions[parts[1]] = {
            "return": parts[0],
            "args": parse_function_args(parts[2]),
            "additional_args": {},
        }
        return
```

```
    if tree_type == "init":
        if len(parts) > 3:
            value = parts[3:]
        else:
            value = None

        functions[list(functions.keys())[-1]]["additional_args"]
[parts[1]] = {
            "type": parts[0],
            "value": value,
        }
        return

    if tree_type == "condition":
        cond = [
            *functions[list(functions.keys())[-1]]["args"],
            *list(functions[list(functions.keys())[-1]]
["additional_args"].keys()),
        ]

        first = parse_tree(parts[0])
        second = parse_tree(parts[2])
        possible_types = list(types.keys())
        if (
            parts[1] in condition_resolver
            and (first in cond or type(first).__name__ in
possible_types)
            and (second in cond or type(second).__name__ in
possible_types)
        ):
            first_temp = functions[list(functions.keys())[-1]]
["args"].get(
                parts[0], False
            ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                parts[0], False
            )
            second_temp = functions[list(functions.keys())[-1]]
["args"].get(
                parts[2], False
            ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                parts[2], False
            )

            if not first_temp:
                first_temp = type(first).__name__
            else:
                first_temp = first_temp["type"]

            if not second_temp:
                second_temp = type(second).__name__
```

```
            else:
                second_temp = second_temp["type"]

            if first_temp == second_temp:
                return

        raise TypeError("Wrong operand types in condition")

    if tree_type == "var_call":
        cond = [
            *functions[list(functions.keys())[-1]]["args"],
            *list(functions[list(functions.keys())[-1]]
["additional_args"].keys()),
        ]
        if parts[0] in cond and (parts[1] in cond or
parts[1].type == "arg"):
            arg = parse_tree(parts[1])
            if arg in cond or isinstance(arg, int):
                return arg
            else:
                raise ValueError(f"Forbidden argument type in
call {arg}")
        else:
            raise NameError(f"Unknown variable name {parts[0]}
or {parts[1]}")

    if tree_type == "func_call":
        function = parts[0]
        if function == "cout":
            output_value = parse_tree(parts[1].parts[1])
            try:
                if output_value.type == "var_call":
                    output_type =
functions[list(functions.keys())[-1]]["args"].get(
                        output_value.parts[0], False
                    ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                        output_value.parts[0], False
                    )
                    if output_type["type"] in ["int", "char",
"string"]:
                        return
                    else:
                        raise ValueError(
                            f"Output operator can't display
value {output_type}"
                        )
            except AttributeError:
                if (
                    isinstance(output_value, (int, float, str))
                    or output_value == "endl"
                ):
                    return
```

```
        elif function in functions.keys():
            arguments = parts[1].parts

            if len(arguments) != len(functions[function]
["args"]):
                raise Exception(
                    f"Wrong count of arguments passing to
function {function}"
                )

            for index, arg in enumerate(arguments):
                try:
                    arg = parse_tree(arg.parts[1])
                except Exception:
                    arg = parse_tree(arg)

                argument = get_type(arg)

                if (
                    argument
                    != functions[function]["args"][
                        list(functions[function]["args"].keys())
[index]
                    ]["type"]
                ):
                    raise ValueError(
                        f"Wrong argument type passing to
function {function}"
                    )
            return

    if tree_type == "assign":
        assign_arguments = []
        for part in parts:
            if part == "=":
                continue

            try:
                if part.type == "var_call":
                    argument = get_type(part.parts[0])

                    assign_arguments.append(argument)
            except AttributeError:
                arg = parse_tree(part)
                argument = get_type(arg)

                assign_arguments.append(argument)

        if not (
            len(assign_arguments) == 2
            and assign_arguments[0] == assign_arguments[1]
            or len(assign_arguments) == 1
        ):
```

```
            raise ValueError(
                f"Can't convert {assign_arguments[1]} to
{assign_arguments[0]}"
            )

    if tree_type == "modal_function":
        if parts[0] == "return":
            argument = parse_tree(parts[1])
            try:
                if eval(functions[list(functions.keys())[-1]]
["return"]) == type(
                    argument
                ):
                    return
                else:
                    raise ValueError("Incorrect return value
from function")
            except Exception:
                raise ValueError(
                    "Using return statement in function than
return 'void'"
                )

        return

    if tree_type == "arg":
        arg = parts[0]
        try:
            if arg.type == "var_call":
                return arg
        except Exception:
            pass

        if isinstance(arg, int):
            return arg
        elif isinstance(arg, float):
            return arg
        elif len(parts) == 1 and re.match(r"(\".*\")|(\'.*\')",
arg):
            return arg
        return

    if tree_type in operations:
        first = parse_tree(parts[0])
        second = parse_tree(parts[1])
        if type(first) != type(second):
            raise TypeError(
                "Types mismatch: {0} and
{1}".format(type(first), type(second))
            )
        if tree_type == "/" and second == 0:
            raise ZeroDivisionError("Unacceptable operation:
division by zero")
```

```
        return operation_resolver[tree_type](first, second)

    for part in parts:
        if part != "=":
            parse_tree(part)


def get_type(value):
    argument = functions[list(functions.keys())[-1]]
["args"].get(
        value, False
    ) or functions[list(functions.keys())[-1]]
["additional_args"].get(value, False)
    if not argument:
        argument = type(value).__name__
    else:
        argument = argument["type"]

    return argument


def check_inits():
    for func in functions:
        additional_args = functions[func]["additional_args"]
        for key, value in additional_args.items():
            value_type = value["value"]
            try:
                if value_type[0].type == "var_call":
                    argument = get_type(value_type[0].parts[0])
                elif value_type[0].type == "arg":
                    arg = parse_tree(value_type[0])
                    argument = get_type(arg)
            except (AttributeError, TypeError):
                pass

            if value_type is not None and argument !=
value["type"]:
                raise ValueError(
                    f"Wrong initialization of variable: type
{argument} can't be equal {value['value']}"
                )


if __name__ == "__main__":
    tree = build_tree(data)
    parse_tree(tree)

    if not functions.get("main", False):
        raise Exception(
            "Program should have starting point as function with
name 'main'"
        )
```

```
check_inits()
print(tree)
```

Приложение 2. Код анализируемой программы

```
void reverseArray(int arr[], int start, int end)
{
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   {
       cout << arr[i];
       cout << " ";
   }
   cout << endl;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);

    reverseArray(arr, 0, n - 1);

    cout << "Reversed array is";
    cout << endl;

    printArray(arr, n);

    int a = 10;
    int b = 10 - 14;
    if (a < b)
    {
        int t = 10;
    }
    else {
        int t = 15;
    }

    cout << t;

    return 0;
}
```