

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. 953501
Корневский С. А.

Проверил:
ст. преподаватель Шиманский В. В.

Минск 2022

Содержание

| | |
|--|----|
| Содержание | 2 |
| 1 Постановка задачи..... | 3 |
| 2 Теория | 4 |
| 3 Результат работы анализатора | 6 |
| 4 Выводы | 9 |
| Приложение А. - Исходный код анализируемой программы | 10 |
| Приложение Б. - Исходный код анализатора | 11 |

1 Постановка задачи

В данной работе ставится задача исследования области семантических ошибок, подробному изучению теории семантики: приведение типов, операции с различными типами. Основной целью работы является семантика, то есть возможность интерпретатора распознавать типы. Это является следующим шагом анализа текста программы – семантический, существенно отличающийся от двух предыдущих – лексического и синтаксического. Таким образом, программа выполнит 3 фазу – выполнение приведения типов и затронет некоторый функционал, связанный с работой интерпретатора

2 Теория

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а содержательными методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со структурными, т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над внутренними объектами, с которыми имеет дело программа. Для начала перечислим все, что их касается и лежит на поверхности:

- большинство объектов являются именованными. Имя объекта позволяет его идентифицировать, существуют различные области действия имен, соглашения об именах, различные умолчания и т.п. Все это относится к семантике;
- виды, сложность и набор характеристик объектов различаются в разных языках программирования и сильно зависят от области приложения языка (в этом смысле семантика языков программирования более разнообразна, нежели синтаксис и лексика).
- объекты связаны между собой (ссылаются друг на друга).

Семантика программы – внутренняя модель множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей.

Теперь, когда у нас есть представление о синтаксической фазе, можно оценить ее центральную роль в организации процесса трансляции.

Лексические единицы независимы друг от друга и являются терминальными символами синтаксиса. Семантика программы тоже не

обладает структурной целостностью и представлена фрагментарно, но при этом связана с синтаксисом следующим образом:

- один и тот же семантический объект (например, переменная) может встречаться в различных, синтаксически несвязанных частях программы;
- синтаксические конструкции описаний, определений и объявлений являются источником семантики объектов программы, они «заявляют» о существовании объектов и задают их свойства;
- синтаксические конструкции, связанные с действиями, выполняемыми над объектами, являются потребителями семантики, их интерпретация, корректность, «смысл» зависят от семантических свойств объекта. Забегая вперед, можно заметить, что заключительная фаза трансляции (генерация кода, интерпретация) может рассматриваться как особые семантические действия, производимые над объектами;
- первичным источником семантики является лексический анализ. Значением лексемы является сама распознанная цепочка литер, она и представляет семантическую составляющую лексемы, которая и обрабатывается;
- лексемы, или то же самое, что терминальные символы входной строки (в терминах синтаксического анализа), ссылаются в семантические таблицы на свою семантику. Формирование семантической составляющей связано с движением снизу-вверх по синтаксическому дереву, от вершин – потомков к предкам.

Таким образом, семантическая составляющая транслятора тоже является фрагментарной (набор семантических процедур, соответствующих правилам грамматики) и объединяется в единое целое только в рамках синтаксического дерева.

3 Результат работы анализатора

Чтобы продемонстрировать работу семантического анализа был приведён код, где показаны переменные и их тип (Рис. 1.)

| +++++ | | |
|-----------|-------|--|
| Variable | Type | |
| +++++ | | |
| i | int | |
| ----- | | |
| j | int | |
| ----- | | |
| mess | str | |
| ----- | | |
| mergeSort | arr | |
| ----- | | |
| SortAlgo | arr | |
| ----- | | |
| L | arr | |
| ----- | | |
| R | arr | |
| ----- | | |
| merged | arr | |
| ----- | | |
| data | arr | |
| ----- | | |
| lenD | int | |
| ----- | | |
| rem | int | |
| ----- | | |
| middle | float | |
| ----- | | |

Рис. 1. Слева указано имя переменной, справа- тип

Из рисунка видно, что последний анализатор способен распознать тип и как следствие появляется возможность отловить последний тип ошибок – семантических, то есть сложение строк с числами, деление на 0

и многих других. Исходя из этих возможностей, можно сделать вывод о заключающем этапе и приближению к программе-интерпретатору.

Код с ошибками:

Изменим исходный код программы, сознательно сделав несколько семантических ошибок.

```
Semantic ERROR (сложение строки и числа)
row 33 pos 8
mes+j
      ...
|      j      |      int      |
-----
|      mess   |      str      |
-----

Process finished with exit code 0
```

Рис. 2. Семантическая ошибка с попыткой сложения строки и числа

```
Semantic ERROR (умножение строки и числа)
row 19 pos 9
mess*lenD
      ...
|      lenD   |      int      |
-----
|      mess   |      str      |
-----
```

Рис. 3. Семантическая ошибка с попыткой умножения строки и числа

```

Semantic ERROR (инкрементация строки)
row 45 pos 3
  mess++

      ...
|    mess    |    str    |
-----

```

Рис. 4. Семантическая ошибка попытка инкрементации строки

```

Semantic ERROR (конкатенация float и int)
row 54 pos 8
  strcat(middle,rem);

      ...
|    rem    |    int    |
-----
|  middle  |    float   |
-----

```

Рис. 5. Семантическая ошибка попытка конкатенации float и int

4 Выводы

Я провёл семантический анализ выбранного языка, сформировано его подмножество, придуман способ, который и способен отлавливать ошибки между разными типами и показаны 4 отловленные ошибки в коде программы на языке C++, успешно отлавливаемые синтаксическим анализатором.

Сложность заключалась в создании отдельного словаря, куда и записываются все переменные с их значениями, которые язык программирования Python, благодаря своей динамической типизации, способен отловить ошибку в преобразованиях, а программа только выводит её..

Семантический анализатор представляет собой третью фазу компилятора, основная задача которого состоит в создании блока программы, решающей проблему с типами и работой их в коде. В дальнейшем это станет необходимым для создания виртуальной машины или, другими словами, интерпретатора.

Язык разбора Python — общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, в то же время включает большой объём полезных функций. Основные архитектурные черты — полнота , простота , поддержка и удобные высокоуровневые инструменты.

Приложение А. - Исходный код анализируемой программы

Данный листинг с кодом демонстрирует исходный анализируемый код с различными типами, такие как унарные и бинарные операции, вывод в консоль, присваивание и т.д.

Исследуемый код языка представлен ниже:

```
void mergeSort(int data[], int lenD)
{
    if (lenD>1) {
        int middle = lenD/2;
        int rem = lenD-middle;
        int *L = (int) middle;
        int *R = (int) rem;
        for (int i=0;i<lenD;i++){
            if (i<middle){
                L[i] = data[i];
            }
            else {
                R[i-middle] = data[i];
            }
        }
        mergeSort(L,middle);
        mergeSort(R,rem);
        merge(data, lenD, L, middle, R, rem);
    }
}
```

```
void merge(int merged[], int lenD, int L[], int lenL, int R[], int lenR){
    int i = 0;
    int j = 0;
    mess = "Done";
    while(i<lenL | j<lenR){
        if (i<lenL & j<lenR){
            if (L[i]<=R[j]){
                merged[i+j] == L[i];
                i++;
            }
            else {
                merged[i+j] = R[j];
                j++;
            }
        }
    }
}
```

```

}
else if (i<lenL){
    merged[i+j] == L[i];
    i++;
}
else if (j<lenR){
    merged [ i+j ] = R[ j ];
    j++;
}
}
}
}

```

Приложение Б. - Исходный код анализатора

Данный листинг с кодом демонстрирует методы с различными типами, такие как унарные и бинарные операции, вывод в консоль, присваивание и т.д.

```

def get_quote(self, skip_spaces=True):
    """
    return quote
    """
    lexem = self.char
    if skip_spaces:
        self.skip_space()
    self.next_char()
    while True:
        if self.char == '(':
            lexem += self.get_quote(False)
        if self.char == ')':
            lexem += self.char
            self.next_char()
        if self.char != ')':
            lexem += self.char
            self.next_char()
        else:
            return lexem
def skip_comments(self, char):
    """
    skip comments
    """
    while self.char != char:
        self.next_char()
    self.next_char()

```

```

        return self.next_token()
def gettoken(self):
    """
    return token
    """
    self.next_char()
    while True:
        result = self.next_token()
        if not result:
            continue
        if result.value == 'EOF':
            break
        yield result
def tokens(self):
    """
    return list of tokens
    """
    result = [i for i in self.gettoken()]
    return result
def raw_input(self, user_string):
    """
    return raw user input
    """
    self.string = user_string
    return self.tokens()
class Parser(object):
    """
    class Parser
    """
    def __init__(self):
        self.tokens = None
    def _node(self, pos):
        """
        return new node and pos
        """
        node = list()
        while self.tokens[pos].value != ')':
            if self.tokens[pos].value == '(':
                new_node, pos = self._node(pos + 1)
                node.append(new_node)
            else:
                node.append(self.tokens[pos])
            pos += 1
        return node, pos
    def build(self, tokens):
        """

```

```

    return ast
    """
    ast = list()
    if tokens:
        pos = 0
        self.tokens = tokens
        while pos < len(tokens):
            if tokens[pos].value == '(':
                node, pos = self._node(pos + 1)
                pos += 1
                ast.append(node)
            else:
                msg = 'Parser error! Expected "(" but given "%s" % tokens[pos].value
                msg += ' in line {}, column {}'.format(tokens[pos].col - 1, tokens[pos].row)
                raise Exception(msg)
        return ast
def obs(env, fun, *args):
    """
    obs
    """
    result = execute(args[0], env)
    for i in args[1:]:
        result = fun(result, execute(i, env))
    return result
def defun(env, *args):
    """
    defune new function
    """
    name, params, *body = args
    proc = Procedure(params, *body)
    if not name.value in env:
        env[name.value] = proc
    else:
        msg = 'Function "%s" already exists!' % name.value
        msg += ' in line {}, column {}'.format(name.col, name.row)
        raise Exception(msg)
def compare(env, *args):
    """
    get condition and execute first or second body
    """
    if execute(args[0], env):
        return execute(args[1], env)
    elif len(args) == 3:
        return execute(args[2], env)
def write(env, *args):
    """

```

```

write line
"""

from sys import stdout
stdout.write(str(execute(args[0], env)))
stdout.flush()
def write_line(env, *args):
    """

    write new line
    """

    from sys import stdout
    stdout.write('%s\n' % str(execute(args[0], env)))
    stdout.flush()
def readint(env, *args):
    """

    read line
    """

    i = 0
    env[args[i].value] = int(input())
    from sys import stdout
    if isinstance(args[i].value, str):
        stdout.write(str(execute(args[0], env)))
        stdout.flush()
def setq(env, *args):
    """

    define new variables
    """

    i = 0
    while i < len(args):
        env[args[i].value] = execute(args[i + 1], env)
        i += 2
def execute(expr, env):
    """

    execute
    """

    if isinstance(expr, Token):
        if expr.tag == ID and expr.value in env:
            return env[expr.value]
        else:
            return expr.value
    else:
        first, *second = expr
        if first.value in env and callable(env[first.value]):
            return env[first.value](env, *second)
        else:
            msg = 'Function "%s" not exists!' % first.value
            msg += 'in line {}, column {}'.format(first.col, first.row)

```

```

        raise Exception(msg)
RESERVED = 'RESERVED'
UNKNOWN = 'UNKNOWN'
NUMBER = 'NUMBER'
STRING = 'STRING'
QUOTE = 'QUOTE'
ID = 'ID'
class Token:
    """
    docstring for Token
    """
    def __init__(self, value, tag, row, col):
        self.value = value
        self.tag = tag
        self.row = row
        self.col = col
    def __str__(self):
        return '<{}, {}, {}, {}>'.format(self.value, self.tag, self.row, self.col)
    def __repr__(self):
        return self.__str__()
class Lexer(dict):
    """
    docstring for Lexer
    """
    def __init__(self, file, *args):
        super().__init__(*args)
        self.pos, self.row, self.col = 0, 1, 1
        self.char = ""
        self.file = open(file, 'r')
        self.string = self.file.readline()
        self.errors_list = list()
    def errors(self):
        """
        print all errors
        """
        import sys
        self.file.close()
        sys.stderr.write('Lexer errors:\n')
        for i in self.errors_list:
            sys.stderr.write('\t%s\n' % i)
        sys.stderr.flush()
        exit(1)
    def error(self, text):
        """
        print error
        """

```

```

self.errors_list.append(
    '{} in line {}, column {}'.format(text, self.row, self.col))
def next_char(self):
    """
    set next char
    """
    if self.pos < len(self.string):
        self.char = self.string[self.pos]
        if self.char != '\n':
            self.col += 1
            self.pos += 1
        else:
            self.string = self.file.readline()
            self.col = 1
            self.row += 1
            self.pos = 0
    else:
        self.char = '#0'
def skip_space(self):
    """
    skip spaces
    """
    while self.char.isspace():
        self.next_char()
def next_token(self):
    """
    return token
    """
    self.skip_space()
    lexem = ""
    # if current char is alpha or _
    if self.char.isalpha() or self.char == '_' or self.char in '+-*/%>=<^!?:':
        lexem = self.char
        self.next_char()
        # adding all alpha and digit
        while self.char.isalpha() or self.char.isdigit() or self.char in '+-*/%>=<^!?:':
            lexem += self.char
            self.next_char()
        return Token(lexem, ID, self.col, self.row)
    # if current char is digit
    elif self.char.isdigit():
        # while is digit
        count = 0
        while self.char.isdigit() or self.char == '.':
            if self.char == '.':
                count += 1

```



```

        if count > 1:
            self.error('Incorrect format of number: "%s"' % self.char)
            lexem += self.char
            self.next_char()
        return Token(int(lexem) if count == 0 else float(lexem), NUMBER, self.col, self.row)
    elif self.char in ('(', ')'):
        lexem = self.char
        self.next_char()
        return Token(lexem, RESERVED, self.col, self.row)
    elif self.char == '#0':
        return Token('EOF', None, self.col, self.row)
    elif self.char == '-.':
        lexem = self.char
        self.next_char()
        if self.char.isdigit():
            count = 0
            while self.char.isdigit() or self.char == '.':
                if self.char == '.':
                    count += 1
                if count > 1:
                    self.error('Incorrect format of number: "%s"' %
                               self.char)
                lexem += self.char
                self.next_char()
            return Token(int(lexem) if count == 0 else float(lexem), NUMBER, self.col, self.row)
    elif self.char in (';', '{'):
        # skip comments in file
        return self.skip_comments('\n' if self.char == ';' else '}')
    elif self.char == '"':
        self.next_char()
        while self.char != '"':
            if self.char == '\\':
                lexem += self.char
                self.next_char()
                lexem += self.char
                self.next_char()
                continue
            lexem += self.char
            self.next_char()
        self.next_char()
        return Token(lexem, STRING, self.col, self.row)
    elif self.char == "'":
        lexem = self.get_quote()
        return Token(lexem[1:], QUOTE, self.col, self.row)
    elif self.char in self:
        lexem = self.char

```

```
        self.next_char()
        return Token(lexem, self[lexem], self.col, self.row)
    else:
        lexem = self.char
        self.error('Unknown character: "%s"' % self.char)
        self.next_char()
        return Token(lexem, UNKNOWN, self.col, self.row)
    return None
```