Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н. Э. Баумана»

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ ПО ДИСЦИПЛИНЕ

«Конструирование компиляторов»

НА ТЕМУ:

«Проверка вложения языков образцов»

Студент ИУ9-72Б _____ Шевляков Е. Д.

Руководитель КР _____ Непейвода А. Н.

Москва 2021 г.

Содержание

Введение

Задача работы состоит в построении алгоритма, в частности, используемого для нахождения экранируемых предложений в программах на языке Рефал. В дополнение к самому алгоритму необходимо написать программу-интерфейс, позволяющую переводить программу на Рефале в множество программ-задач для алгоритма проверки вложения образцов.

В первой части данной записки описывается предметная область и ставится задача. Во второй части реализуется поставленная задача. В третьей части описывается процесс тестирования разработанных программ.

1 Описание предметной области

Рассмотрим некоторые моменты языка Рефал, которые имеют значимость для написания требуемого алгоритма и его понимания, определим, что имеется в виду под экранированием предложений и обозрим теорию языков образцов, на которой основан алгоритм.

1.1 Рефал

РЕФАЛ (Рекурсивных Функций Алгоритмический язык) – это функциональный язык программирования, ориентированный на так называемые "символьные преобразования": обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом.

Программа, написанная на Рефале, состоит из функций. Каждая функция, в свою очередь, состоит из заглавия и тела, в котором описан ряд предложений в фигурных скобках. Под предложением понимается пара (образец, выражение), разделенная знаком равенства: =.

Работа функции заключается в сопоставлении ее аргумента (у функций Рефала всегда один аргумент) с образцами: сначала аргумент сопоставляется с образцом первого предложения. Если сопоставление дало положительный результат, то вычисляется выражение этого предложения (значение, полученное в результате вычисления выражения, является итогом работы функции). Если же сопоставление прошло неудачно, то те же самые действия проходят на следующем предложении, и так далее. Если аргумент не удалось сопоставить ни с каким образцом из предложений функции, то возникает ошибка применения функции. Пример функции Рефала представлен на листинге 1.

```
Palindrom {
    s.1 e.2 s.1 = <Palindrom e.2>;
    s.1 = True;
    /* пусто */ = True;
    e.1 = False;
}
```

Листинг 1. Пример функции Рефал

Как видно на рисунке 1, образец представляет собой некую последовательность, элементы которой разделены пробельными символами, не заключенными в кавычки.

Рассмотрим все типы элементов, из которых могут состоять образцы:

- Символы – последовательность знаков, которая не распознается как переменная.
- Переменный символ – Обозначается как s.x, где x есть некая последовательность букв, цифр или знаков. Переменная этого типа может принимать значение любого символа.
- Терм – Обозначается как как t.x, где x есть некая последовательность букв, цифр или знаков. Терм может принимать значение любого символа либо любого объектного выражения (Объектное выражение есть выражение, состоящее из символов и структурных скобок ( ). Скобки должны образовывать правильную скобочную структуру. Выражение может быть, в частности, и пустым. Знаки "пробел", не входящие в последовательность простых символов, служат для внешнего оформления программы.
- Переменные типа e.x – отвечают последовательности термов (в том числе и пустой).

1.2 Экранирование предложений

Мы рассмотрели ключевые моменты теории для постановки проблемы. Используя вышеописанные структуры, возможно составить различные образцы, под каждый из которых подошло бы одно и то же константное выражение. Более того, возможно написание таких образцов P_1 и P_2, что множество выражений, сопоставимых с P_2, будет подмножеством множества выражений, сопоставимых с P_1. Если в функции в предложении A образцом будет P_1, а в предложении B будет P_2, то если A стоит раньше B – предложение B никогда не выполнится. Такой случай называется экранированием предложений.

Для примера рассмотрим функцию на листинг 2.

```
ExampleFunction {
    e.1 A = True;
    B s.1 A = False;
    t.1 e.1 t.1 b e.2 = 2;
    t.2 a t.1 b e.3 = 3;
}
```

Листинг 2. Функция с экранированием предложений

Здесь второе предложение является частным случаем первого.

Экранирование предложений происходит по ошибке автора кода. Дело в том, что образцы бывают значительно более замысловатые, чем в примере на рисунке 2, и программисту нетрудно будет не заметить факт экранирования. Такого рода ошибки тормозят разработку ПО и негативно сказываются на читаемости кода и на скорости работы функций (при наличии в них экранирования) за счет холостых проходов при неудачах в сопоставлении.

1.3 Теория языков образцов

Рассмотрим теорию языков образцов, на которой основан алгоритм, реализованный для решения вышеописанной проблемы:

- V – множество переменных
- $\sum$ – достаточно большой алфавит констант

Плоский образец – строка в алфавие $V \cup \sum$

Образец (вне зависимости от ого, плоский он или нет) линеен, если кратность каждой e-переменной в нем равна 1.

Подстановка σ – морфизм из $(V \cup \sum)^*$ в $(V \cup \sum)^*$, сохраняющий константы (т. е. Для всех $A \in \Sigma\ \sigma(A) = A$).

Не плоские образцы, содержащие повторные t-переменные, в данной работе не рассматриваются.

Языком L(P), распознаваемым образцом P (англ. – pattern language, сокращенно PL), называется множество элементов $\Phi \in \Sigma^*$, для которых существует подстановка σ: σ(P) = Φ. Образец P_1 сводится к образцу P_2, если $L(P_1) \subseteq L(P_2)$.

Если для e-переменных допустима пустая подстановка то PL называют стирающим (англ. – erasing, сокращенно EPL). Если недопустимо – не стирающим (сокращенно NePL).

- EPL, распознаваемый образцом-строкой $P \in \Sigma$, есть {P}.
- EPL, распознаваемый образцом $P = e.x_1\ e.x_2\ .\ .\ .\ e.x_n$, есть все множество Σ . NePL, распознаваемый этим образцом — множество слов, имеющих не меньше, чем n букв.

Дан образец P и служебные символы ◁ и ▷. Якорем называется подслово α образца ◁ P ▷ такое, что:

- α не содержит e-переменных
- Любое надслово, входящее в P, содержит e-переменную

В образце может быть больше чем один якорь. Последовательность якорей, входящих в P, записываем через ||.

Например, образец $e.x_1$ A B $e.x_2$ $e.x_3$ C соответствует последовательности якорей ◁‖AB‖C▷.

Теорема: Наличие s-переменных не изменяет свойств языков образцов. (считаем, что алфавит может содержать структурные скобки, т.е. тип данных, используемый по умолчанию — выражение).

Якорная t-переменная – t-переменная, удовлеворяющая одному из условий:

- t имеет кратность, не меньшую 2
- в P существует подслово α, не содержащее e-переменных, такое, что α = α_1 t.x α_2 , причем α_1 и α_2 оба содержат хотя бы один символ или t-переменную, имеющую кратность не меньше 2.

Теорема (о сводимости образцов): Если образцы P_1 и P_2 линейные и все их t-переменные якорные, то P_1 сводится к P_2 ⇔ существует подстановка σ такая, что $\sigma(P_2) = P_1$.

Если в образце появляются плавающие t-переменные, многие полезные свойства линейных образцов не выполняются.

Плавающая переменная в образце – указатель на то, что в соответствующий фрагмент образца нельзя подставить пустое слово.

Плавающий сегмент образца P – подслово P, содержащее хотя бы одну плавающую t-переменную и e-переменную, и ничего, кроме плавающих t-переменных или e-переменных.

Образец, в котором все e-переменные входят в плавающие сегменты – аналог образца, определяющего NePL. Достаточно заменить e-переменные на новый тип переменных, отличных от e-типа тем, что исключают возможность пустой подстановки(назовем такие переменные v-переменными), и пользоваться теорией NePL для определения вложения языков. Если это верно только для некоторых e-переменных — получаем образец, определяющий смешанный язык – не NePL, но и не EPL.

Линейный образец P находится в нормальной форме, если

1. P не содержит двух или более идущих подряд e-переменных
2. Каждая плавающая t-переменная в P предшествует и следует за e-переменной

Теорема: Всякому линейному образцу P соответствует (с точностью до переименовки переменных) ровно один образец P′ в нормальной форме такой, что L (P) = L (P′).

Переменная s.x (t.x) перекрещивается с плавающим сегментом F образца P, если образец P можно представить в виде P_1 s.x P_2 F P_3 s.x P_4. Подстроки P*i* произвольны и сами могут содержать вхождения переменной s.x.

Теорема (о сводимости образцов в нормальной форме): Для любых двух линейных образцов P_1 и P_2 в нормальной форме, таких, что в образце P_2 нет переменных, перекрещивающихся с плавающими сегментами, если L (P_1) ⊆ L (P_2), то существует подстановка σ такая, что σ(P_2) = P_1.

Другими словами, если линейный образец P представляется в виде $P_1 P_2 . . . P_n$, где каждый из P_i не содержит одновременно e- и v-переменных, и множества переменных из образцов P_i и P_j для всех $i \neq j$ не пересекаются, то экранирование образцом P можно проверять с помощью проверки на существование подстановки.

Алгоритм для проверки вложения EPL языков: Пусть P_1, P_2 таковы, что P_1 линейный, плоский и не содержит плавающих t-переменных. Задача — проверить включение $L(P_2) \subseteq L(P_1)$. Пусть N — длина максимального подслова P_1, содержащего только t-переменные. Построим подстановку σ такую, что $\sigma(e.x_i) = (A_{i1}) . . . (A_{iN+1})$, $\sigma(t.x_i) = (B_i)$, $\sigma(s.x_i) = C_i$, где все A_{ij}, B_i, C_i — различные константы , и пользуемся алгоритмом сопоставления константного выражения с образцом. $L(P_2) \subseteq L(P_1)$ тогда и только тогда, когда $\sigma(P_2) \in L(P_1)$.

Алгоритм для проверки вложения NePL языков: Пусть P_1, P_2 линейные, плоские, и вхождение каждой e-переменной в них соседствует с вхождением плавающей t-переменной. Дополнительно требуем, чтобы повторные вхождения t и s-переменных в P_1 не разделялись вхождением e-переменной. Задача — проверить включение $L(P_2) \subseteq L(P_1)$. Заменяем в P_2 каждую плавающую t-переменную на свежую v-переменную, а все e-переменные стираем. Пусть N — длина максимального подслова P_1, принадлежащего якорю и содержащего только t-переменные. Построим подстановку σ такую, что $\sigma(v.x_i) = (A_{i1})...(A_{iN+1})$, $\sigma(t.x_i) = (B_i)$, $\sigma(s.x_i) = C_i$, где все A_{ij}, B_i, C_i – различные константы , и пользуемся обычным алгоритмом сопоставления константного выражения с образцом. $L(P_2) \subseteq L(P_1)$ тогда и только тогда, когда $\sigma(P_2) \in L(P_1)$.

2 Реализация

Основываясь на теории образцов, я сформировал следующий алгоритм, который и является предложенным мной решением поставленной задачи:

В первую очередь, каждый образец из входной пары представляется как последовательность примитивов. В качестве примитива выступает класс Atom, продемонстрированный на листинге 3.

```
class Atom:
    """Элемент образца"""

    def __init__(self, val):
        self.val = val
        self.type = self._get_type(val)

    @staticmethod
    def _get_type(val):
        if val.startswith('e.'):
            return 'e'
        elif val.startswith('t.'):
            return 't'
        elif val.startswith('s.'):
            return 's'
        else:
            return ‘c’
```

Листинг 3. Код атома

Далее происходит ряд действий, определяющих вид образцов, в соответствии с чем, выбирается метод, которым будут сравниваться образцы для получения вердикта. Схема определения метода сопоставления изображена на рисунке 1.

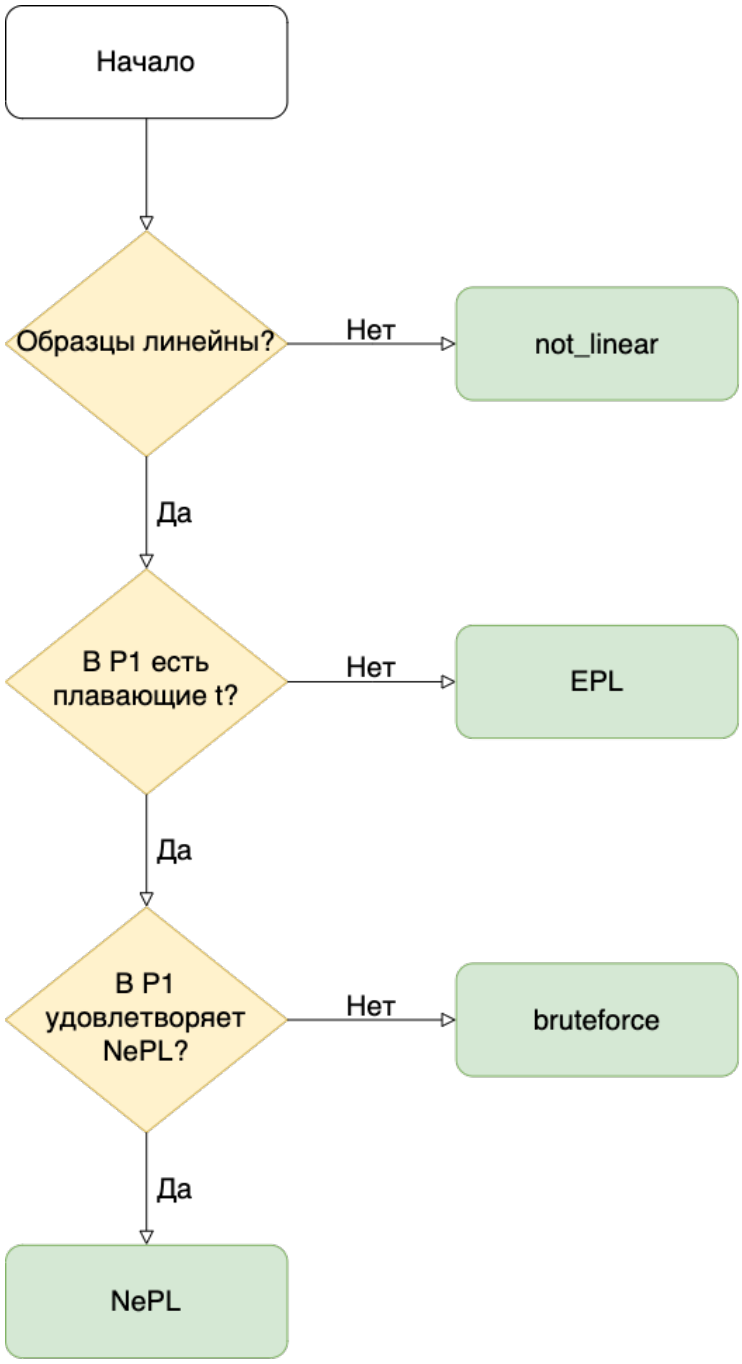

Рисунок 1. Схема выбора метода сопоставления в P_1

Алгоритм проверки вложения EPL:

1. Нормализация образца P_1.
2. Нахождение длины крупнейшего подслова P_1, состоящего только из t-переменных.
3. Построение из P_2 результата константной подстановки.
4. Сопоставление константной подстановки с образцом P_1.

Алгоритм проверки вложения NePL:

1. Нормализация образца P_1.
2. Нахождение длины крупнейшего подслова P_1, состоящего только из t-переменных.
3. Замена e-переменных v-переменными.
4. Построение из P_2 результата константной подстановки.
5. Сопоставление константной подстановки с образцом P_1.

Алгоритм проверки вложения Brute Force:

1. Замена s-переменных в P_2 на свежие константы.
2. Нахождение подстановок в разделённые переменные образца P_1.
3. Нахождение подстановок в e-переменные образца P_2 для каждого варианта из пункта 2.
4. Проверка, является ли множество возможных подстановок в e-переменные универсальным.

Алгоритм проверки вложения Not linear: Сопоставление образцов напрямую. Образцы с t-переменными не рассматриваются.

Рассмотрим более подробно алгоритм полного перебора (Brute force).

Перед пунктом 2 применяется оптимизация: копия P_1 преобразуется (разделенные переменные окружаются e-переменными, затем, из полученного

образца исключаются плавающие переменные). Проверяется вложение P_2 в измененный P_1. Если вложения нет, то экранирования нет.

В пункте 3 для каждой подстановки в разделенные переменные выполняется алгоритм усеченного сопоставления, который дает на выходе множество подстановок в e-переменные. Затем все такие множества объединяются в одно.

Чтобы проверить, универсально ли полученное в пункте 3 множество, применяется алгоритм упрощения, основанный на условиях на длины переменных. После работы алгоритма усеченного сопоставления элементом множества подстановок является композиция пар «e-переменная + сопоставленная ей последовательность». Теперь каждая пара заменяется на объект, характеризующий минимальную длину подстановки и возможность роста этой длины. Для этого используется новый примитив SubAtom, реализация которого представлена на листинге 4.

```
class SubAtom:
    def __init__(self, val, power, generate=False):
        self.val = val
        if generate:
            self.len, self.have_plus = SubAtom._build(power)
        else:
            self.len, self.have_plus = power[0], power[1]
    @staticmethod
    def _build(atoms):
        e_exist, tf_cnt = False, 0
        for atom in atoms:
            if atom.type == 'e':
                e_exist = True
            elif atom.type == 'tf':
                tf_cnt += 1
        return tf_cnt, e_exist
```

Листинг 4. Атом-подстановка

Также, была разработана простая программа для составления заданий для алгоритма из кода на языке Рефал. Программа находит функции с помощью регулярного выражения «\S+ \\{[^\\}]+\\}» и части предложений в них с помощью регулярного выражения «'\'[^\']*\'». Программа рассчитана на функции, состоящие из названия и тела, в котором присутствуют только предложения. Весь функционал программы – в файле work_creator.py.

3 Тестирование

3.1 Юнит-тесты

Процесс разработки сопровождался написанием юнит-тестов. Для этого использовался модуль unittest. Все такие тесты описаны в файле unit_tests.py.

Рассмотрим класс EnclosureTest, часть кода которого представлена на листинге 5.

```
class EnclosureTest(unittest.TestCase):

    def __init__(self, method):
        super().__init__()
        self.method = method

    def work_tests(self, exs):
        for ex in exs:
            with self.subTest(i=ex):
                self.assertEqual(
                    self.method(ex.ps[0], ex.ps[1]),
                    ex.answer, msg=ex.ps)

    ...
```

Листинг 5. Код класса EnclosureTest.

Данный класс использовался для испытания алгоритма на различных типах пар образцов, для которых уже известен правильный ответ. Для каждого типа образцов создавался отдельный метод. Рассмотрим для примера метод, отвечающий за проверку на образцах с кратными t-переменными, код которого представлен на листинге 6.

```
def with_repeated_t(self):
    """Проверка на образцах с кратными t-переменными"""
    examples = [
        Example(
            ('e.y0 t.y e.y1 t.y1 t.y C e.y2',
             'C A e.x1 A B C e.x2'), False),
        Example(
            ('t.1 A e.2 t.2 t.1',
             'B A t.x e.3 B'), True),
        Example(
            ('t.1 A t.2 e.1 t.1',
             'B A e.x t.3 B A'), False),
        Example(
            ('e.1 t.x e.2 t.2 t.x e.3',
             'e.1 t.1 t.1 e.2 t.2 t.3 t.4 t.1 e.3'), True),
    ]
    self.work_tests(examples)
```

Листинг 6. Метод класса EnclosureTest.

Как видно на листинге, создается список тестов в виде специальных структур Example, который затем последовательно запускается экземпляром класса. Остальные методы класса EnclosureTest построены аналогичным образом. Этот класс позволил исправлять старые ошибки алгоритма, избегая появления новых.

3.2 Тестирование на программах

Следующим этапом тестирования стало тестирование на программах, написанных на языке Рефал. Программы, находящиеся в каталоге «refal examples», состоят из последовательностей функций. Пример такой программы на листинге 7.

```
Check_0 {
    <e.x1 'AAB' e.x2 e.x3> = <RHS>;
    <e.x1 'A' 'B' e.x2> = <RHS>;
    <e.x1  'AB' e.x2> = <RHS>;
    <e.x1 s.z1 e.x2 e.x3> = <RHS>;
    <e.x1  e.x2> = <RHS>;
}

Check_1 {
    <e.x1 e.x2 'AAB' e.x3> = <RHS>;
    <e.x1 e.x2 s.z1 e.x3> = <RHS>;
    <'AB' e.x1 e.x2> = <RHS>;
    <'B' e.x1> = <RHS>;
    <s.z1 e.x1 e.x2> = <RHS>;
    <e.x1> = <RHS>;
}
```

Листинг 7. Пример входной программы.

При данном тестировании для каждой программы данного каталога использовался функционал из ранее описанного файла work_creator.py. В результате чего, программа разбивалась на функции, в функции находились образцы, из образцов формировались пары и подавались на вход алгоритму.

Для каждой программы формировался csv-файл с результатами работы. Пример такого файла на листинге 8.

```
function,pattern1,pattern2,result
Check_0,e.x1 'AAB' e.x2 e.x3,e.x1 'A' 'B' e.x2,False
Check_0,e.x1 'AAB' e.x2 e.x3,e.x1  'AB' e.x2,False
Check_0,e.x1 'AAB' e.x2 e.x3,e.x1 s.z1 e.x2 e.x3,False
Check_0,e.x1 'AAB' e.x2 e.x3,e.x1  e.x2,False
Check_0,e.x1 'A' 'B' e.x2,e.x1  'AB' e.x2,False
Check_0,e.x1 'A' 'B' e.x2,e.x1 s.z1 e.x2 e.x3,False
Check_0,e.x1 'A' 'B' e.x2,e.x1  e.x2,False
Check_0,e.x1  'AB' e.x2,e.x1 s.z1 e.x2 e.x3,False
Check_0,e.x1  'AB' e.x2,e.x1  e.x2,False
Check_0,e.x1 s.z1 e.x2 e.x3,e.x1  e.x2,False
Check_1,e.x1 e.x2 'AAB' e.x3,e.x1 e.x2 s.z1 e.x3,False
Check_1,e.x1 e.x2 'AAB' e.x3,'AB' e.x1 e.x2,False
Check_1,e.x1 e.x2 'AAB' e.x3,'B' e.x1,False
Check_1,e.x1 e.x2 'AAB' e.x3,s.z1 e.x1 e.x2,False
Check_1,e.x1 e.x2 'AAB' e.x3,e.x1,False
Check_1,e.x1 e.x2 s.z1 e.x3,'AB' e.x1 e.x2,True
Check_1,e.x1 e.x2 s.z1 e.x3,'B' e.x1,True
Check_1,e.x1 e.x2 s.z1 e.x3,s.z1 e.x1 e.x2,True
Check_1,e.x1 e.x2 s.z1 e.x3,e.x1,False
Check_1,'AB' e.x1 e.x2,'B' e.x1,False
Check_1,'AB' e.x1 e.x2,s.z1 e.x1 e.x2,False
Check_1,'AB' e.x1 e.x2,e.x1,False
Check_1,'B' e.x1,s.z1 e.x1 e.x2,False
Check_1,'B' e.x1,e.x1,False
Check_1,s.z1 e.x1 e.x2,e.x1,False
```

Листинг 8. Пример отчета.

3.3 Журналирование

Для отслеживания выполнения этапов программы ведется используется журналирование. Всего было создано два уровня журналирования. Каждый для своего способа тестирования.

Первый уровень журналирования используется для создания детализированной отчетности прохождения пары входных образцов через опорные ключевые «точки» алгоритма. Этот уровень удобен при тестировании небольшого количества пар образцов. Пример вывода данного журналирования на листинге 7.

```
DEBUG:root:Test: [t.1, A, e.2, t.2, t.1][B, A, t.x, e.3, B]
DEBUG:root:BruteForce: [t.1, A, e.2, t.2, t.1][B, A, t.x, e.0, B]
DEBUG:root:Splited vals:{t.1}
DEBUG:root:EPL: [e.0, t.1, e.1, A, e.2, t.1, e.3][B, A, t.x, e.0, B]
DEBUG:root:SplitSubstitution: [[B, A, e.2, t.2, B]]
DEBUG:root:
ESubstitution:
((e.2, t.2),)
((),)
DEBUG:root:Sets of SubAtoms:[{e.0:1+}, {e.0:0}]
DEBUG:root:Simplification: [set()]
```

Листинг 7. Журналирование первого уровня

Второй уровень журналирования используется для создания отчетности о порядке проведения тестов. Этот уровень удобен при тестировании большого количества пар образцов, когда детализированная информация окажется чрезмерно объемной. В этом случае удобно отследить конкретный участок тестов, который потом можно рассмотреть, используя журналирование первого уровня. Пример вывода данного журналирования на листинге 8.

```
INFO:root:Start with refal examples/test1.ref
INFO:root:Work function IterEncode
INFO:root:Work function GenBen
INFO:root:Work function CopyFile
INFO:root:Work function Conversion
INFO:root:Work function ConvertConst
INFO:root:Start with refal examples/test2.ref
INFO:root:Work function AlphabetT
INFO:root:Work function IfInSetMultiple
INFO:root:Work function Translate
INFO:root:Work function Eq
```

Листинг 8. Журналирование второго уровня

Заключение

В ходе работы был реализован алгоритм проверки вложения языков образцов и программа-интерфейс, переводящую простые программы на языке Рефал в задачи для разработанного алгоритма. Таким образом, поставленные цели были достигнуты.

Список источников

1. Содружество «Рефал/Суперкомпиляция» URL:http://www.refal.net/ (Дата обращения 12.2020)
2. РПЗ Барлука Выдача предупреждений для экранируемых предложений 2020
3. Angluin, D.: Finding Patterns Common to a Set of Strings. In Journal of Computer and System Sciences. 21: 46–62 (1980)
4. Shinohara, T.: Polynomial time inference of pattern languages and its applications, Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science, 191–209, (1982)
5. Jiang, T., Salomaa, A., Salomaa, K. and Yu, S.: Inclusion is undecidable for pattern languages, In Proceedings of the 20th International Colloquium, ICALP’93, Lecture Notes in Computer Science, Springer-Verlag, 301–312, (1993)
6. Kari, L., Mateescua, A., Punb, G., Salomaaa, A.: Multi-pattern languages. In Theoretical Computer Science 141, 253-268 (1995)
7. Reidenbach, D.: Discontinuities in pattern inference. In Theoretical Computer Science 397, 166–193 (2008)